Un modèle génératif pour le développement de serveurs Internet

Gautier Loyauté

UNIVERSITÉ PARIS-EST

THÈSE DE DOCTORAT

pour l'obtention du grade de

Docteur de l'Université Paris-Est

Spécialité Informatique

au titre de l'École Doctorale Information, Communication, Modélisation et Simulation

Présentée et soutenue publiquement par

Gautier LOYAUTÉ

le 5 Septembre 2008

Un modèle génératif pour le développement de serveurs Internet

Sous la direction de :

M. Gilles ROUSSEL

Devant le jury composé par:

Rapporteurs :	Mme Laurence DUCHIEN
	M. Didier PARIGOT
Examinateurs :	M. Rémi FORAX
	M. Gilles ROUSSEL
	M. Pierluigi SAN PIETRO

Remerciements

L'écriture des remerciements en seulement quelques mots est l'un des exercices les plus difficiles qui soit, car il s'agit d'écrire et de décrire ce qui peut parfois sembler convenu.

Il me semble important de souligner le rôle du directeur de thèse qui est la personne clef dans la conduite d'un doctorat. C'est pourquoi, je tiens en tout premier lieu à exprimer mes remerciements à Gilles Roussel qui a accepté de m'encadrer durant ce long chemin. Avec sa propension à répondre à toutes mes questions, son aide apportée pour la rédaction des différents articles et pour cette présentation à Lisbonne. Ses questions semblant parfois anodines qui en y réfléchissant bien, éclairaient sous un angle nouveau les travaux ou idées. Enfin les discussions diverses et variées échangées ainsi que ses commentaires m'ont permis de rendre un manuscrit un peu plus présentable.

Je remercie Laurence Duchien et Didier Parigot de m'avoir fait l'honneur de rapporter ma thèse, pour leurs remarques pertinentes et les conseils qui m'ont permis d'améliorer ce document.

Je remercie très vivement Pierluigi San Pietro pour avoir accepté de faire partie du jury et de l'intérêt qu'il a bien voulu porter à mon travail.

Je remercie Rémi Forax qui a toujours d'excellentes idées et une connaissance « encyclopédique » du langage Java ou d'applications s'y rapportant, ainsi que Julien Cervelle pour l'aide apportée sur Tatoo et le soutien sans faille face à la *furia* de certains étudiants.

Je remercie Etienne Duris pour l'aide qu'il m'a apportée concernant aussi bien des questions de Java que de réseau, mais aussi son soutien, ses conseils en tant que tuteur pédagogique de monitorat et pour avoir accepté de relire les pré-versions de ce manuscrit.

Je tiens à remercier Nicolas Bedon, qui fut mon directeur de stage de maîtrise, pour l'aide et les conseils apportés tout au long de ce parcours. Plus particulièrement, pour ses encouragements et son soutien avant chaque entrée dans « l'arène aux fauves » ou durant cette période si difficile qu'est la rédaction d'une thèse.

Je tiens à souligner qu'à mes yeux cette thèse fut avant tout constituée de rencontres et de personnes qui ont compté et comptent encore... A tous, je souhaitais dédier cette thèse et leur exprimer ma reconnaissance.

Résumé

Les serveurs Internet sont des logiciels qui présentent des caractéristiques particulières car ils doivent répondre aux demandes d'un grand nombre de clients distants, supporter l'évolution du nombre de clients qui peut être brusque et importante et être robuste car ils ne doivent jamais s'arrêter.

Les modèles de concurrence permettent d'entrelacer les traitements d'un grand nombre de clients. Leur variété tient à l'utilisation de concepts de programmation différents (entrées/sorties, processus) et aux divergences d'organisation du code, cependant aucun consensus ne se dégage sur un meilleur modèle. Pour s'abstraire du modèle de concurrence, je propose dans cette thèse un modèle de développement de serveurs Internet. Il produit automatiquement par génération le code concurrent.

Les outils de vérification formelle permettent d'accroître la sûreté des logiciels. Toutefois, il est nécessaire de fournir un modèle simple du logiciel pour rendre possible la vérification. Le modèle de développement de serveurs que je propose est utilisé pour générer automatiquement le serveur et son modèle formel. Ce qui permet d'augmenter la sûreté du modèle vis-à-vis de l'application qu'il modélise.

Enfin comme la lecture/décodage d'une requête cliente dépend du modèle de concurrence je propose d'utiliser Tatoo un générateur d'analyseur syntaxique. Tatoo s'abstrait de ce problème et automatise le développement du décodage. Des analyseurs indépendants du type des E/S sont générés en fonction du protocole.

Ces contributions ont donné lieu à une implantation et intégration dans Saburo une « fabrique » de serveurs Internet en Java.

Mots clés : Serveurs Internet, modèle de concurrence, modèle de développement, analyseur syntaxique, modèle formel, HTTP, Java, NIO.

Abstract

Internet servers are softwares with specific features. Indeed, they answer requests of wide and distant clients, support the customer evolution and must be robust as they never stop.

The concurrency models allow to interleave the statements of wide customers. Different concepts of programming (inputs / outputs and process) and various organizations of code imply a large panel of models. However, no general agreement frees on a better model. To abstract the concurrency model, I propose a development model of Internet servers. My model produces automatically concurrency code using a generation approach.

Model chekers allow to increase the software safety. Nevertheless, it is necessary to provide a simple model of software to check it. I propose to use my development model of servers in order to generate the server automatically and its formal model. This allows to increase the model safety in relation with the application which it models.

Finally, as the request parsing depends on the concurrency model, I propose to use Tatoo, a parser generator. Tatoo abstracts himself from this dependency and automatizes the development of parsing. Independent parsers of the I/O type are generated according to the protocol.

These contributions rise to an implementation in Saburo, an Internet servers factory in Java.

Keywords : Internet servers, concurrency model, software engineering, compiler, formal model, HTTP, Java, NIO.

Table des matières

Table des figures

1.

Introduction et motivation

L'explosion d'Internet durant cette dernière décennie est marquée par le déploiement sur une grande échelle d'un très grand nombre de services [27]. Pas seulement dominés par des sites Web au contenu statique, les services Internet sont maintenant aussi divers que des sites marchands en ligne (Amazon), des messageries instantanées (*MicroSoft Network Messenger*), du partage pair-à-pair de fichiers (Bittorent), de la diffusion de contenu multimédia (radio en ligne) ou enfin, de l'hébergement d'applications (*Application Service Provider*). Contrairement aux sites à contenu statique, ces nouveaux types de services ont recours à des calculs significatifs côté serveur ainsi qu'à d'importantes opérations d'entrées / sorties (E/S) pour répondre à la requête d'un client. De plus pour délivrer ces nouveaux services, d'autres systèmes sont requis pour stocker des données (bases de données), accélérer les transactions (caches), assurer la confidentialité des données (services d'authentification) ou enfin, faciliter l'accès, la configuration et l'utilisation de ces services (interfaces Web). Parallèlement, ces nouveaux services doivent aussi faire face à l'augmentation importante des utilisateurs d'Internet en étant robustes, c'est-à-dire exempts de « comportements non désirés », et en agissant correctement aux demandes de ces très nombreux clients, qui sont potentiellement des millions !

Les services Internet sont devenus de plus en plus importants aussi bien pour les industriels que pour les particuliers. Ainsi, les industriels utilisent des applications Internet pour le commerce en ligne (e-commerce), la gestion des chaînes de production ou celle des ressources humaines (candidatures en ligne). De même, beaucoup de particuliers considèrent la messagerie électronique et les accès au Web comme devenus indispensables. Cette « dépendance » met en exergue les propriétés de disponibilité, de montée en charge et de capacité à supporter longtemps des charges importantes des différents services Internet. En effet, certains sites très populaires présentent, lors des pics d'accès, des ralentissements importants et dérangeants pour leurs utilisateurs. Ainsi en 2001, durant une semaine, le service MSN (*MicroSoft Network*) a subi des ralentissements conséquents qui ont entraîné des déconnections intempestives du service de messagerie instantanée [110].

Plus le nombre de personnes connectées à Internet continuera à croître, plus les services

Internet devront avoir des comportements exemplaires et robustes face aux variations fréquentes et brusques de la charge.

Ainsi parmi les services les plus populaires, les sites d'information en ligne sont sujets, en fonction de leurs contenus et de l'actualité, à de très fortes variations du nombre de visiteurs pouvant parfois atteindre des facteurs d'ordre 20 en quelques minutes [64].

Les services Internet sont fournis par des *serveurs logiciels* qui tentent de satisfaire les demandes des différents clients. Le développement d'un serveur Internet doit donc prendre en compte ce problème sans précédent qu'est le *support d'un grand nombre d'utilisateurs accédant simultanément à un seul service*. En plus de cette concurrence massive, le développement d'un serveur Internet doit tenter de résoudre les problèmes de *robustesse face à la charge*, d'*hébergement sur des machines hétérogènes*, de *génération de contenu dynamique* ou enfin, d'*évolution des fonctionnalités fournies*. De plus, les modèles de concurrence vont avoir des coûts plus ou moins importants en ressources. Par exemple, un serveur Internet peut être utilisé pour configurer un pda ou répondre aux requêtes de milliers de clients sur une machine multi-processeurs. Les ressources disponibles et utilisées vont alors être extrêmement variables. Pour toutes ces raisons, le développement d'un serveur Internet n'est pas simple et requiert beaucoup d'investissement humain et financier.

1.1 Problématique

Techniquement, les différentes actions (connexions, recherches, calculs, etc.) concurrentes effectuées sur un serveur Internet vont se traduire invariablement par des opérations d'E/S sur des interfaces réseaux et sur le disque dur ainsi que des calculs sur la machine d'hébergement. Afin d'entrelacer les traitements (E/S et calculs) des différentes requêtes clientes concurrentes, on utilise traditionnellement des processus ou des processus légers. Cependant, cette approche implique un coût important en terme d'empreinte mémoire, en temps d'ordonnancement ou en nombre de bascules du processeur entre les différents processus à exécuter [1, 41, 85]. Pratiquement, on peut remarquer une baisse des performances des serveurs Internet qui utilisent un modèle créant et attribuant un processus léger par connexion entrante (voir Fig. 1.1). Dans ce modèle, plus le nombre de clients va augmenter et plus le nombre de processus légers va augmenter. Jusqu'à un certain seuil les performances de ces serveurs vont croître mais, passer ce seuil...

Une autre approche consiste à utiliser des E/S non bloquantes mais, ce type d'E/S est difficile à utiliser car il est nécessaire de gérer manuellement la sauvegarde des contextes [1, 11, 12, 41]. De plus, il existe actuellement plusieurs architectures matérielles possibles : (i) machines mono-processeur, (ii) machines multi-processeurs, (iii) clusters, (iv) pda, etc. ainsi que plusieurs techniques d'entrelacements des traitements de requêtes concurrentes, appelées *modèles de concurrence*. Pratiquement, les modèles de concurrence vont avoir des coûts plus ou moins importants en ressources. Ainsi, un serveur Internet peut être utilisé pour configurer un pda *via* une connexion réseau ou tenter de répondre aux requêtes de milliers de

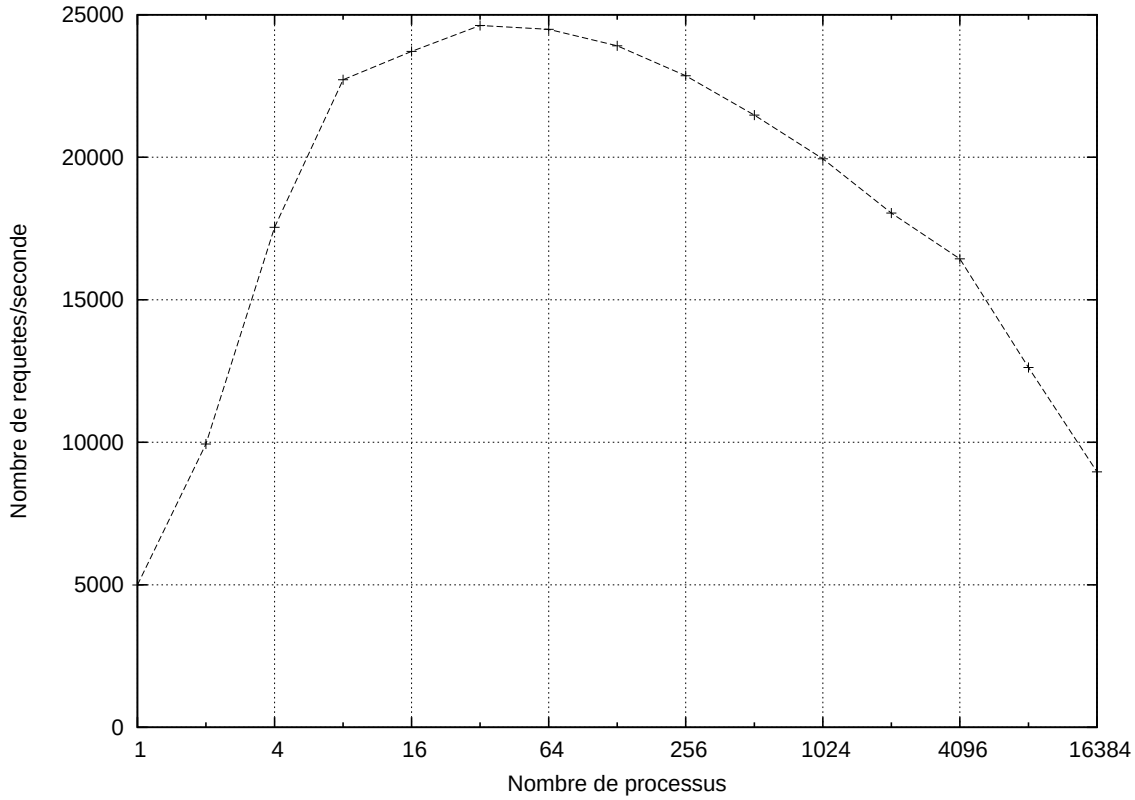


FIG. 1.1 – Variation des performances en fonction du nombre de processus légers.

clients sur une machine multi-processeurs. Les ressources disponibles vont alors être extrêmement diverses et le serveur Internet devra s'adapter aux besoins des clients et aux ressources disponibles. Pour chacune des architectures matérielles existantes, il existe un modèle de concurrence donné permettant d'exhiber les meilleures performances possibles et qui répond exactement aux demandes du développeur. Comme les serveurs Internet sont hébergés sur des machines hétérogènes, il est donc nécessaire d'adapter le plus facilement et rapidemment possible le modèle de concurrence en fonction de l'architecture matérielle sous-jacente afin de satisfaire le maximum de client en un minimum de temps et les besoins du développeur.

Cependant, la plupart des travaux de recherche sur les serveurs Internet se focalise sur l'optimisation des performances [12, 86, 88, 114]. Du fait des nombreuses optimisations de code, de l'imbrication de la partie « métier » et des différentes préoccupations (dont la concurrence) le code source des serveurs Internet est difficilement lisible, maintenable et évolutif. C'est pourquoi pour adapter le modèle de concurrence à chaque architecture matérielle sous-jacente, il est nécessaire de redévelopper de « bout en bout » le même serveur Internet. Il y a donc un véritable manque de rationalisation en temps et en coût de développement de ce type d'application [109, 111].

1.2 Contribution

Durant ma thèse de doctorat, j'ai tenté de résoudre le problème de rationalisation du développement des serveurs Internet en proposant une architecture de « fabrique » de serveurs Internet. Par extension naturelle cela permet également d'augmenter la sûreté des serveurs ainsi produits.

1.2.1 Saburo, une fabrique de serveurs Internet

L'idée centrale développée durant ma thèse de doctorat est l'application du principe de *séparation des préoccupations* [50] pour faciliter le développement de serveurs Internet.

Pour atteindre ce but, j'ai tout d'abord cherché à identifier les principales caractéristiques de développement de chaque modèle de concurrence existant. Pour se faire, j'ai proposé une classification des modèles de concurrence (voir Fig. 1.2 et section 2.3) pour en faciliter l'étude [69]. Il existe d'autres classifications des modèles de concurrence [1, 86, 114], mais celle que je propose est plus simple et se base sur deux critères que je considère comme prépondérants dans la spécification d'un modèle de concurrence :

- le type d'E/S (bloquantes ou non-bloquantes) ;
- l'utilisation des processus (un seul processus, coopération ou compétition).

En effet, l'utilisation de ces critères m'a permis de m'abstraire de toutes les variantes implantatoires ou optimisations de chacun de ces modèles.

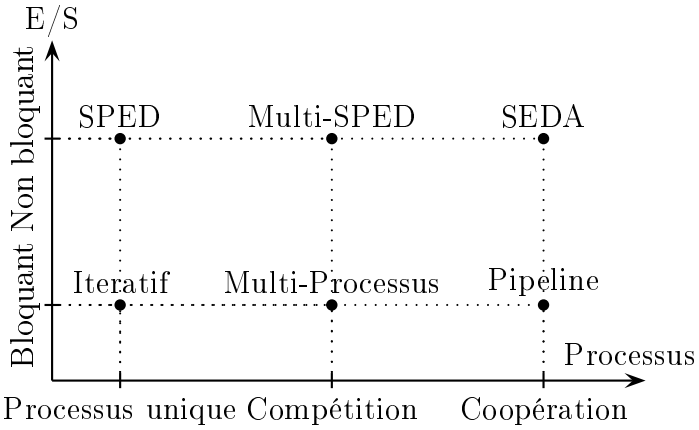


FIG. 1.2 – Classification des modèles de concurrence

Puis en étudiant différentes implantations [7, 86, 109, 114] de ces modèles de concurrence, j'ai réussi à extraire des « patrons de conception » de concurrence pour chacun de ces modèles (voir section 3.1.2). En effet, j'ai remarqué que le code « métier » du serveur ne va pas dépendre du modèle de concurrence choisi contrairement à la structure du code, du mode des E/S et de l'utilisation des processus [68, 70].

Définition 1.1 *Patron de conception*

Les patrons de conception décrivent des solutions standards pour répondre à des problèmes d'architecture et de conception des logiciels.

J'ai ensuite proposé un modèle de spécification de serveurs Internet qui me permet de spécifier un serveur Internet sous la forme d'un graphe orienté (voir Fig. 1.3 et section 3.1.3).

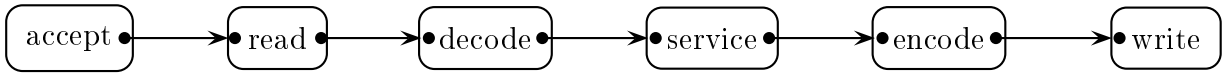


FIG. 1.3 – Graphe de synchronisation et de communication d'un serveur HTTP

Ce graphe représente l'ensemble des opérations de synchronisation présent dans le serveur Internet à développer. Les nœuds, appelés *stage* en référence à l'architecture SEDA [109] (*Staged Event-Driven Architecture*), vont représenter une suite d'instructions fournissant un traitement fondamental du serveur (E/S sur une connexion réseau, accueil d'un nouveau client, etc.). Les deux caractéristiques très importantes de ces bouts de code sont :

- l'absence de toute synchronisation (représentée par le graphe) ;
- une seule opération d'E/S possible dans le code d'un stage.

Enfin à l'aide des générateurs de code qui sont fournis par ma « fabrique » et qui génèrent le code dépendant du modèle de concurrence choisi, je suis capable de tisser le code « métier », ici les stages, et le code concurrent pour obtenir un serveur Internet pleinement fonctionnel [68, 70]. L'opération de *tissage* a pour but d'entrelacer automatiquement le code métier et le code de concurrence dans le but de produire le serveur.

L'originalité de mon approche est qu'à partir d'une spécification déclarative de la structure d'un serveur Internet je suis capable de *générer le code source d'un « aspect » global d'un serveur Internet : son modèle de concurrence*. Cette approche similaire à la programmation par aspects, *spécifique à mes besoins*, a le mérite d'être *mise en œuvre très simplement*, par une extension naturelle du patron de conception visiteur [37] (mes générateurs) sur la spécification abstraite du serveur Internet (le graphe orienté, les stages et le modèle de concurrence). Plus prosaïquement et vue l'absence de consensus sur le meilleur modèle de concurrence [11, 41, 85], son intérêt est de *passer très facilement et de manière transparente d'un modèle de concurrence à un autre* en fonction des besoins et de la plateforme. Elle offre une *diminution du temps de développement* et permet aux développeurs de se consacrer à l'enrichissement du service fourni, à l'ajout de fonctionnalités ou à l'amélioration de la sûreté du serveur. Elle permet aussi d'*éviter un grand nombre d'erreurs* de programmation et d'*améliorer la sûreté* des serveurs Internet ainsi obtenus car le code concurrent, traditionnellement sujet à de nombreux comportements non désirés, est produit automatiquement. De plus, le code obtenu par génération est un *code dédié* qui offre de meilleures performances qu'un code générique. Enfin, mon approche est *évolutive* car, par simple ajout de générateurs, il est possible de prendre en compte de nouveaux couples architectures matérielles - modèles de concurrences.

J'ai illustré ma méthode de développement en réalisant un prototype de « fabrique » de serveurs Internet en Java, nommé Saburo [68, 70, 71] (présentée dans la section 3.2). Saburo est une bibliothèque Java qui permet de déclarer le graphe de spécification et les stages d'un serveur Internet. Elle permet aussi de fournir des classes d'encapsulation permettant de simplifier l'utilisation de l'API NIO [104] qui fournit des E/S bloquantes et non bloquantes mais qui est complexe d'utilisation. Elle offre des générateurs de code pour les différents modèles de concurrence que j'ai définis précédemment (voir Fig. 1.2) ainsi que quelques stages de base réutilisables « sur étagères ». Enfin, elle fournit une interface XML (*eXtensible Markup Language*) basée sur l'outils Ant [6] pour simplifier la spécification et la génération automatique d'un serveur Internet.

1.2.2 Vérification automatique de serveurs Internet

Afin d'augmenter la sûreté d'une application, il existe des outils de vérification automatique, appelés *model checkers*. Cependant, leur utilisation reste bien souvent cantonnée au monde académique ou aux applications industrielles sensibles (logiciel de gestion du cœur des centrales nucléaires d'EDF par exemple) du fait d'un formalisme perçu comme difficile. De plus, lors de la vérification automatique d'un système, ces méthodes vont réaliser une exploration exhaustive, i.e. combinatoire, de tous les états de la représentation formelle de celui-ci pour tester et déverminer toutes les situations possibles.

Plus le système va être complexe et plus le nombre d'états de sa représentation formelle va être important !

Pour résoudre ce problème d'explosion du nombre d'états, il est nécessaire d'écrire une abstraction d'un programme pour qu'il soit facilement vérifiable par un outil de vérification automatique.

L'écriture d'une bonne abstraction est importante et difficile car il est nécessaire de réaliser un compromis entre le nombre d'états dans l'abstraction et l'intérêt des résultats obtenus lors de la vérification. En particulier l'abstraction doit être *sûre* vis-à-vis des propriétés à vérifier, c'est-à-dire que la présence d'une erreur dans le programme original doit également être retrouvée dans l'abstraction. Actuellement, l'abstraction est souvent spécifiée à la main ce qui peut introduire un biais entre le modèle et l'application qu'il modélise, i.e. l'abstraction n'est plus sûre vis-à-vis du programme original (voir Fig. 1.4).

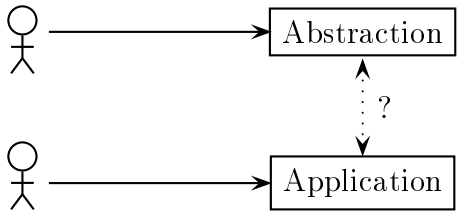


FIG. 1.4 – Abstraction spécifiée manuellement

Je pense qu'il est préférable que l'abstraction soit obtenue automatiquement à partir du système à vérifier pour garantir la sûreté de l'abstraction vis-à-vis de l'application.

Il existe des logiciels générant automatiquement des abstractions sûres (voir Fig. 1.5) pour n'importe quel programme C ou Java [4, 28, 44]. Ces approches générales nécessitent de spécifier formellement des règles d'extraction de l'abstraction en fonction des différentes propriétés que l'on souhaite vérifier.

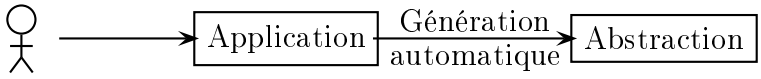


FIG. 1.5 – Abstraction obtenue automatiquement à partir de l'application

Je pense que devoir spécificier formellement ces différentes règles nuit à l'utilisation de tels outils !

Plus généralement, je pense que proposer des générateurs d'abstractions spécifiques à des domaines précis, tels les serveurs Internet [71], les applications distribuées [57] ou les interfaces graphiques [14], évite la spécification des propriétés que l'on souhaite vérifier car elles dépendent la plupart du temps du domaine, simplifie le développement des générateurs et améliore la sûreté des abstractions vis-à-vis du système.

C'est pourquoi, j'ai réutilisé le graphe de spécification d'un serveur Internet (voir section 1.2.1) ainsi que les modèles de concurrence comme langage d'entrée d'un générateur automatique d'abstraction sûre. Le code des différents stages n'est pas nécessaire pour obtenir l'abstraction du serveur Internet car l'ensemble des opérations de synchronisation et de communication présentes dans un serveur Internet sont modélisées *via* ce graphe de spécification. Ces informations vont être suffisantes pour la vérification des propriétés principales d'un serveur Internet que sont :

- l'absence d'interblocage ;
- l'atteignabilité de tous les états du système.

Similairement à la génération automatique du serveur, la génération automatique d'abstraction sûre va utiliser des « patrons de conception » associés à chaque modèle de concurrence. Ces patrons de conception vont permettre de modéliser les intéractions entre les stages ainsi que le type d'utilisation des processus dans l'abstraction. Enfin, il faut noter que l'abstraction et le serveur Internet fonctionnel sont obtenus automatiquement à partir d'une même spécification (voir Fig. 1.6).

Cette approche permet d'*obtenir automatiquement les abstractions* sans avoir à spécifier les propriétés à vérifier mais, pour accroître la sûreté de l'application, il est possible d'*ajouter des propriétés supplémentaires* à vérifier, sous forme de formules de logique temporelle. La

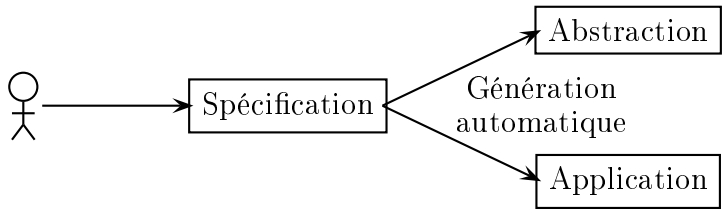


FIG. 1.6 – Abstraction et application obtenues automatiquement d'une unique spécification

sûreté de l'abstraction vis-à-vis de l'application qu'elle modélise est garantie par les différents générateurs que je fournis. Enfin, cette approche descendante [14, 57, 71] permet de *simplifier considérablement le développement des générateurs d'abstraction* comparativement à des méthodes ascendantes [4, 28, 44]. En effet, le langage d'entrée, dans mon cas le graphe de spécification du serveur et le modèle de concurrence, est extrêmement simple comparativement à un langage de programmation complet tel que le C ou le Java.

J'ai illustré cette méthode d'obtention automatique d'abstraction sûre pour les serveurs Internet développés à l'aide de Saburo en fournissant des générateurs d'abstraction sûre vers le langage Promela [71], langage d'entrée du *model checker* SPIN [47].

1.2.3 Génération automatique de l'analyse syntaxique

Lors du développement d'un serveur HTTP (*HyperText Transfer Protocol*) simple, je me suis rendu compte que l'implantation de la phase de décodage des requêtes clientes était longue, fastidieuse, sans réelle difficulté technique (le plus dur étant de respecter scrupuleusement le langage de communication entre le serveur et les clients) et surtout extrêmement répétitive !

De plus selon le type des E/S, non bloquantes ou bloquantes, le code se différencie par la sauvegarde ou non du contexte du décodage de la requête. En effet, les E/S non bloquantes se basent sur l'enregistrement de bouts de code qui sont exécutés lorsque survient l'événement d'E/S qui les intéresse. Par exemple, si un événement de lecture se produit, le lecteur enregistré pour cet événement va être réveillé et recevoir le flot de données provenant de l'interface d'E/S. Si le flot de données se tarit, deux cas se présentent :

- soit la fin du flot, marquée explicitement, est atteinte et le lecteur associé est désenregistré ;
- soit il est mis en sommeil en attendant d'être réveillé lorsque de nouvelles données seront disponibles.

Par exemple lors de la lecture d'un mot w, le lecteur peut être arrêté à un préfixe p_w de ce mot car la suite du mot w n'est pas disponible. Après une période d'attente t_w, le lecteur est réveillé car un facteur f_w du mot w est nouvellement disponible (voir Fig. 1.7).

Inversement, dans le cas d'E/S bloquantes, la routine de lecture va extraire les données de l'interface d'E/S. Lorsque le flot de données se tarit, deux cas se présentent :

- soit la fin du flot est atteinte et l'instruction de lecture peut fermer l'interface d'E/S ;

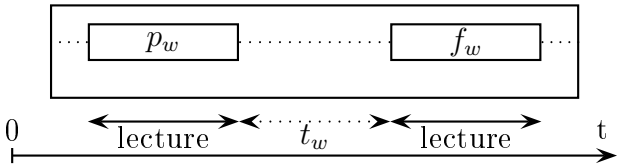


FIG. 1.7 – Lecture non-bloquante d'un mot w

- soit elle attend activement que de nouvelles données soient disponibles.

Par exemple lors de la lecture du mot w, le lecteur peut être arrêté à un préfixe p_w de ce mot car la suite du mot w n'est pas disponible. Pendant une période d'attente t_w, le lecteur va attendre activement qu'un facteur f_w du mot w soit nouvellement disponible (voir Fig. 1.8).

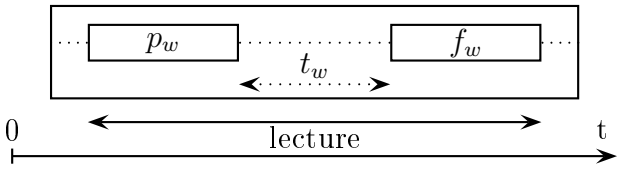


FIG. 1.8 – Lecture bloquante d'un mot w

Comme la principale motivation de ma thèse est de fournir une méthode et des outils qui permettent de passer très facilement et de manière transparente d'un modèle de concurrence à un autre et que j'ai caractérisé les modèles de concurrence en fonction du type de leurs E/S et de leur gestion des processus (section 1.2.1). Il m'est donc nécessaire, pour répondre à mon objectif, de proposer une méthode qui permette de s'abstraire du type des E/S utilisées par le modèle de concurrence. De plus en s'inscrivant dans l'idée d'une fabrique de serveur Internet, je vais utiliser des outils de génération automatique de code source pour automatiser le développement du décodage des requêtes clientes [22].

Pour se faire, je me suis basé sur l'ensemble des règles et des procédures à respecter pour émettre ou recevoir des informations sur le réseau et qui définissent le protocole de communication. De même que pour les langues naturelles, un protocole de communication va être défini par un *dictionnaire*, une *grammaire* et une *sémantique* :

- le *dictionnaire* est l'ensemble des mots du langage. Les *mots* sont des séquences de symboles choisis sur un alphabet ;
- la *grammaire* définit les règles de syntaxe du protocole c'est-à-dire, les règles qui permettent de construire des énoncés corrects. Elle est normative ;
- la *sémantique* donne un sens aux énoncés construits *via* la grammaire mais, dans le cas d'un protocole de communication, elle représente les actions à effectuer en fonction de la requête d'un client.

A partir des descriptions formelles du dictionnaire et de la grammaire d'un protocole de communication, standardisées sous forme de RFC (*Request For Comments*), et à l'aide de Tatoo, un générateur d'analyseur syntaxique [23], j'ai pu obtenir automatiquement le code

analysant les requêtes clientes [22]. Tatoo, contrairement aux principaux générateurs d'analyseurs syntaxiques actuels [36, 60, 90], permet de générer automatiquement des analyseurs lexicaux et syntaxiques compatibles avec des E/S bloquantes et non bloquantes.

J'ai déjà souligné l'intérêt principal de cette approche qui est l'*adaptation automatique de la phase de décodage des requêtes clientes au type des E/S* (bloquantes ou non bloquantes) et donc au modèle de concurrence. Elle offre aussi une *diminution du temps de développement* ce qui permet d'*augmenter la productivité des développeurs*. Elle permet d'*éviter des erreurs de programmation* dues à des erreurs humaines lors du développement de ces portions de code ce qui *augmente la sûreté* du serveur Internet ainsi produit.

Dans le but d'illustrer cette approche, j'ai spécifié les principales caractéristiques nécessaire aux analyseurs syntaxiques que l'on souhaite embarquer au sein de serveurs Internet. Ces besoins (compatibles avec des E/S bloquantes et non bloquantes, gestion mémoire, etc.) ont guidé la réalisation d'un générateur d'analyseurs syntaxique nommé *Tatoo* [23] car aucun générateur d'analyseurs actuel ne répondait à ces contraintes particulières. J'ai ensuite réalisé l'association de Saburo et Tatoo [22] et j'ai développé un serveur HTTP simple (voir section 5.3.2) qui me permet de montrer la faisabilité de ma méthode de développement et de cette association.

1.2.4 Une contribution à la programmation générative

Outre la rationalisation du développement de serveurs Internet et l'amélioration de la sûreté de ces applications *via* la génération automatique de code et l'intégration d'outils de *model checking*, l'intérêt de tous ces travaux est de donner une illustration de la programmation générative pour une famille d'applications bien particulière, les serveurs Internet. En cela, je rejoins [89] et je fournis un exemple, *via* Saburo, de fabrique d'applications qui est l'idée forte de cette approche pour la production d'applications ouvertes et évolutives.

1.2.4.1 Fabrique de logiciels

Je suis convaincu que pour assurer une meilleure évolution et surêté lors du développement de logiciels, il est essentiel qu'une grande partie des applications soit produite automatiquement *via* des générateurs de code. Ces générateurs de code vont systématiser l'application, pour une même famille de logiciels, de patrons de conception ou de préoccupations orthogonales à l'application de base, telle que la persistance ou la concurrence. Cette approche de programmation générative va changer l'application des patrons de conception qui, traditionnellement, est laissée à l'interprétation des développeurs et dont l'implantation est effectuée *a posteriori*, c'est-à-dire lors du développement de l'application. Ici, les divers patrons de conception nécessaires à une famille d'applications vont être identifiés à l'avance et appliqués automatiquement par les générateurs de code. Cette approche permet au développeur de se consacrer exclusivement aux parties « métiers » de son application. De plus, la maintenance et l'évolution des applications sont grandement facilitées car les générateurs vont produire les parties des applications les plus sensibles aux évolutions technologiques.

Il suffit de les modifier pour prendre en compte de nouvelles évolutions ou demandes dans toutes les applications qu'ils ont produites.

1.2.4.2 Modèles dédiés par opposition à modèles génériques

Je suis persuadé de l'intérêt d'une approche par famille d'applications, plutôt qu'une approche basée sur la notion de modèle générique telle que les technologies par composants EJB (*Enterprise JavaBeans*) ou encore l'approche UML (*Unified Modelling Language*). En effet comme d'autres travaux [15, 89], il me semble préférable de définir un modèle adapté au « cœur de métier » sous-jacent car il s'abstrait des aspects techniques ou implantatoires pour mieux identifier les notions spécifiques associées à la famille d'applications, c'est-à-dire à un métier particulier. Saburo est un exemple de cette approche dédié aux serveurs Internet. Plus généralement, je pense que les modèles doivent être abstraits, c'est-à-dire indépendants de toutes technologies, et dédiés à des domaines particuliers plutôt qu'à des modèles génériques universels. En effet, répondre aux évolutions technologiques et aux demandes des utilisateurs très rapidement devient un véritable défi que les modèles universels peuvent difficilement relever.

1.2.4.3 Séparation des préoccupations

Enfin, je suis convaincu que l'approche par séparation des préoccupations est une idée prépondérante pour le futur du développement de logiciels. L'intérêt de cette approche vient d'une idée fondamentale en informatique qui est le *développement incrémental* d'une application, i.e. par étapes successives. La programmation par aspects a donné les moyens d'une telle approche et a répondu aux besoins d'*intégration de fonctionnalités non prévues initialement*, de *retours fréquents entre la phase d'implantation et la phase de spécification*, de *maintenance des applications*, de *modifications successives* et enfin, de *réduction de la complexité des applications et de leurs environnements d'exécution*. Tout cela explique le succès de cette nouvelle thématique qu'est la programmation par séparation des préoccupations, dont la programmation par aspects est l'une des solutions. Je pense que la définition de préoccupations de plus en plus complexes ainsi que le problème de réutilisation d'une préoccupation dans un autre modèle nécessitera le recours à l'ingénierie dirigée par des modèles.

1.3 Organisation du manuscrit

La suite de ce manuscrit se décompose en trois grandes parties. Tout d'abord, je vais présenter Saburo, ma « fabrique » de serveurs Internet, ainsi qu'un exemple de son utilisation *via* le développement d'un serveur HTTP simple. Puis, je vais présenter deux extensions à Saburo permettant d'accroître la sûreté des serveurs Internet produits. La première extension est la proposition d'une méthode descendante de génération automatique d'abstraction sûre de serveurs Internet développés à l'aide de Saburo. La seconde extension est l'intégration de Tatoo, un générateur d'analyseur syntaxique [23], à Saburo, afin de générer automatiquement la phase de décodage des requêtes d'un client. Enfin, je vais détailler les aspects techniques pour développer un serveur Internet performant en Java. Je vais les appliquer

dans un générateur de code de Saburo afin de montrer que l'on peut obtenir un serveur très performant. Je vais enfin comparer ce serveur à d'autres serveurs pour valider pratiquement mon approche.

Le chapitre 2 présente les *processus* et les *processus légers* concepts de base de la concurrence au sein des systèmes informatiques. En effet, dans de nombreuses applications, des activités multiples ont lieu simultanément et ces deux modèles conceptuels ont été introduits afin de faciliter le partage du processeur, le développement et le « déverminage » de ce parallèlisme logiciel. Une autre motivation à l'introduction des processus est la nature bloquante de certaines des activités d'une application (par exemple, la lecture sur une connexion réseau). Lorsqu'une telle activité se produit au sein d'un processus, le processeur peut alors être attribué à un autre processus et son utilisation sera ainsi optimisée dans une application. Néanmoins, il est maintenant de plus en plus courant que la lecture et l'écriture sur les interfaces d'E/S soient des opérations non bloquantes. C'est pourquoi je vais présenter les différents modèles d'E/S existants actuellement. Enfin, je donnerai une classification des différents modèles de concurrence en fonction de deux critères que je trouve prépondérants : (i) le type des E/S et (ii) l'utilisation des processus.

Le chapitre 3 présente mon modèle de développement de serveurs Internet ainsi qu'un prototype Java nommé Saburo. Saburo tente de répondre pratiquement à de nouvelles exigences provenant, pour la plupart, de contraintes liées : (i) au temps de développement, (ii) à son financement, (iii) aux disparités de connaissances des différents utilisateurs ainsi (iv) qu'aux nécessités d'adaptation rapide aux besoins du marché et des matériels. Cependant, du fait de nombreuses optimisations de code, de l'imbrication de la partie « métier » et du modèle de concurrence, le code source des serveurs Internet est difficilement lisible, maintenable et évolutif. Dans un premier temps, je vais présenter un modèle de développement de serveurs Internet qui est basé sur la notion de *fabrique de logiciels* et de *séparation des préoccupations*. Je vais aussi décrire succinctement des concepts et techniques qui sont intégrés dans Saburo à différents niveaux. Je vais ensuite présenter Saburo, un prototype Java. Enfin dans le but d'illustrer l'utilisation de Saburo, je vais détailler le développement d'un serveur HTTP.

Le chapitre 4 décrit l'exploitation de mon modèle de développement pour extraire automatiquement un modèle formel d'un serveur Internet, *via* une méthode descendante. Après avoir présenté les principes généraux des *models checkers*, je vais décrire une méthode d'extraction de modèle formel. Cette extraction se fait à partir de mon modèle de développement. L'intégration des méthodes de vérification formelle directement dans le cycle de développement est un défi important car elles permettent d'accroître fortement la sûreté de l'application et d'en faciliter le déverminage. Cependant, ces outils restent cantonnés à des utilisations ponctuelles, du fait d'un formalisme considéré comme difficile. L'un des principaux intérêts de cette approche est d'obtenir automatiquement et facilement le modèle formel d'un serveur Internet. Enfin, l'illustration de la faisabilité de cette approche est réalisée *via* l'exemple d'extraction de mon modèle de développement vers le langage Promela, langage d'entrée du *model checker* SPIN [47].

Le chapitre 5 présente les détails de l'intégration de Tatoo, un générateur d'analyseur syntaxique, à mon outil Saburo. Après quelques rappels des notions principales de compilation, ce chapitre décrit les caractéristiques principales que doivent présenter les analyseurs syntaxiques à embarquer dans des serveurs Internet. Je vais présenter Tatoo, un générateur d'analyseurs syntaxiques dédié à cette problèmatique d'embarquement dans des applications performantes à longue durée de vie ainsi que son intégration dans Saburo. Les principaux avantages de cette approche est d'automatiser le développement du décodage des requêtes clientes. Mais aussi, d'accroître l'abstraction du code développé par l'utilisateur vis-à-vis des modèles de concurrence. En effet, selon le type d'E/S et donc du modèle de concurrence, la phase d'analyse des requêtes clientes va, ou ne vas pas, sauvegarder le contexte du décodage d'une requête. Tatoo, à la différence des principaux générateurs d'analyseurs syntaxiques actuels, permet d'obtenir des analyseurs qui sont totalement compatibles avec les E/S bloquantes et non-bloquantes. Enfin, l'illustration de la faisabilité de cette méthode est réalisée par le développement d'un serveur HTTP simple.

Le chapitre 6 présente un certain nombre de considérations techniques utiles pour le développement d'un serveur Internet performant en Java. Je vais montrer pratiquement que l'association de Saburo et Tatoo ne nuit en aucun cas aux performances des serveurs développés. Pour se faire je vais comparer les performances de mon serveur HTTP obtenu par mon approche avec celles d'autres serveurs HTTP performants [7, 81, 102, 107].

Enfin, le chapitre 7 donne une conclusion sur l'ensemble de ces travaux et inclut un certain nombre de perspectives pour de futurs travaux.

Première partie

Saburo, une fabrique de serveurs Internet

2.

Concurrence et E/S au sein des serveurs Internet

Sommaire

Dans de nombreuses applications, des activités multiples ont lieu simultanément. Ainsi dans le cas particulier d'un serveur Internet, celui-ci va tenter de répondre aux requêtes envoyées par ses clients. Une première solution est de développer le serveur sans utiliser de mécanismes particuliers pour répondre à cette concurrence. Le serveur va alors récupérer une requête, l'examiner et la mener à son terme avant de passer à la suivante. Cependant, certaines opérations peuvent être bloquantes, par exemple la lecture sur une connexion réseau ou sur un disque dur. Ainsi pendant qu'il attend une réponse du disque, le serveur va être innocupé, c'est-à-dire qu'il ne va traiter aucune autre opération et donc aucune autre requête entrante !

Des mécanismes doivent être mis en place pour traiter « en même temps » le maximum de clients. Plus exactement, il est nécessaire de proposer des outils permettant de maximiser le taux d'occupation du processeur.

C'est pourquoi des modèles conceptuels, basés sur la notion de *processus séquentiels* ont été introduits afin de faciliter la conception, la programmation et le déverminage d'applications concurrentes. Ainsi en décomposant une application en plusieurs processus légers séquentiels qui vont s'exécuter quasiment en parallèle, il est possible de traiter plusieurs tâches « en même temps » et la programmation d'application concurrente devient beaucoup plus simple. En effet, au lieu de penser en termes d'interruptions, de timers ou de changements de contexte, il est possible de raisonner en termes de processus parallèles.

Dans ce contexte précis, un serveur Internet va être constitué de plusieurs processus séquentiels réalisant la même boucle de traitement. Un processus, dédié, va récupérer une nouvelle requête et la transmettre à un processus de traitement. Celui-ci va examiner la requête puis la mener à son terme avant de passer à une autre. Pendant qu'il attend la réponse du disque, le processus de traitement va être bloqué, c'est-à-dire qu'il ne va réaliser aucun traitement. Mais durant cette période d'attente, le processeur va être attribué à un autre processus de traitement par le système d'exploitation sous-jacent. Cette « bascule » permet de traiter un nombre bien supérieur de requêtes comparativement à la première solution présentée.

Cependant, il existe une autre méthode qui exploite les appels systèmes d'E/S non bloquants. Lorsqu'une requête arrive, le seul et unique processus séquentiel en place l'examine. Si celle-ci peut être lue directement tout va bien. Dans le cas contraire, l'opération d'enregistrement de la lecture sur la connexion est déclenchée et le serveur va :

- conserver l'état de la requête en cours pour cette tâche ;
- restaurer l'état d'une requête précédente si de nouvelles données sont disponibles, récupérer les informations adéquates et continuer de traiter cette requête.

Dans ce modèle de conception, le modèle de « processus séquentiel » présenté précédemment est perdu. En effet, l'état du traitement doit être explicitement enregistré et restauré à chaque fois que le serveur bascule d'une requête à une autre. Ce modèle de conception fait intervenir une *machine à nombre d'états finie* car chaque traitement possède un état enregistré dans lequel il existe un jeu d'événements susceptibles de se produire qui permet de passer d'un état à un autre.

Dans ce chapitre, je vais commencer par présenter les processus et les processus légers qui sont les concepts de base de la concurrence au sein des systèmes informatiques actuels. Une classification des différents modèles d'E/S existants va ensuite être établie et décrite. Enfin en conjuguant un modèle d'utilisation des processus et un modèle d'E/S, il est possible d'obtenir différents « modèles de concurrence ». C'est pourquoi la fin de ce chapitre sera dédiée à une classification et une description des principaux modèles de concurrence actuellement existants [69].

2.1 Mécanismes de concurrence

Bien qu'exécutant des programmes utilisateurs, les systèmes d'exploitation actuels sont capables d'effectuer « en même temps » des lectures sur le disque, d'afficher du texte à l'écran ou de recevoir des données d'une connexion réseau. Néanmoins, chaque processeur matériel ne va réellement exécuter qu'une seule tâche à la fois. C'est pourquoi et afin de donner l'impression de simultanéité dans l'exécution des différentes tâches de l'utilisateur et du système d'exploitation, le processeur va mélanger l'exécution de plusieurs instructions de différents programmes. Ainsi, chaque tâche va utiliser le processeur pendant un laps de temps donné. Une fois ce laps de temps terminé, le processeur va réaliser, sous la direction d'un programme d'ordonnancement de tâches, une bascule pour exécuter les instructions

d'un autre programme.

Cependant, il est difficile de concevoir et de « déverminer » plusieurs tâches en parallèle du fait d'un indéterminisme dans leur ordre d'exécution. C'est-à-dire que l'on ne sait ni comment, ni dans quel ordre les opérations des différentes tâches vont s'exécuter et théoriquement toutes les combinaisons sont possibles ! C'est pourquoi, les concepteurs de systèmes d'exploitation ont introduit des modèles conceptuels reposant sur la notion de « processus séquentiels » afin de faciliter la conception, la programmation et le « déverminage » de ce parallélisme logiciel.

2.1.1 Les processus

Dans le modèle de processus, toutes les applications, et parfois même le système d'exploitation, vont être représentés par des *processus séquentiels*. Un *processus séquentiel* ou *processus* va correspondre au flot d'exécution du programme accompagné de ses valeurs de registres et de variables. Ainsi, un processus va conceptuellement avoir son propre processeur. En réalité, le processeur physique va régulièrement « basculer » d'un processus à un autre. La méthode de bascule, c'est-à-dire le choix du moment où arrêter un processus pour en servir un autre, est appelé *ordonnancement*.

Il existe de nombreux algorithmes d'ordonnancement qui ont été conçus pour tenter d'équilibrer les demandes concurrentes, garantir l'efficacité du système dans son ensemble et l'équité entre les processus. Ces algorithmes se divisent en deux catégories :

- *non-préemptifs* : l'ordonnanceur va sélectionner un processus, puis le laisser s'exécuter jusqu'à ce qu'il bloque, soit sur une E/S, soit en attente d'un autre processus ou qu'il libère volontairement le processeur.
- *préemptifs* : l'ordonnanceur va sélectionner un processus et le laisser s'exécuter pendant un délai déterminé, un *quantum de temps*. Si le processus est toujours en cours d'exécution à l'issue de ce délai, il est suspendu, et l'ordonnanceur sélectionne un autre processus à exécuter.

Bien qu'un processus soit une entité totalement indépendante, il arrive régulièrement que certains processus soient bloqués en attente d'entrées non disponibles (voir Fig. 2.1). En effet, un processus peut générer une sortie utilisée en entrée par un autre processus. Il se peut également qu'un processus prêt à s'exécuter soit interrompu car l'ordonnanceur à décider d'attribuer le processeur à un autre processus.

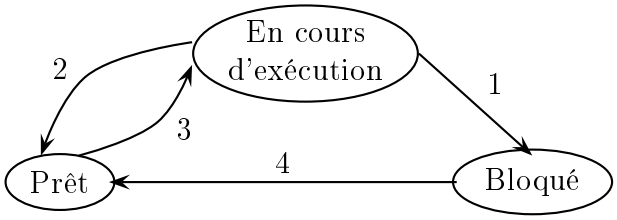


FIG. 2.1 – Les différents états d'un processus

Les différents états que peut prendre un processus sont (voir Fig. 2.1) :
- les états *prêt* et *en cours d'exécution* qui sont analogues. Mais dans le premier cas, le processeur est provisoirement indisponible ;
- l'état *bloqué* qui est différent des deux précédents car le processus ne peut s'exécuter même si le processeur est libre.

Les transitions entre les états, i.e. la bascule d'un état à un autre, sont (voir Fig. 2.1) :
- la *transition* 1 qui se produit lorsqu'un processus ne peut plus poursuivre son exécution, c'est-à-dire qu'il lui manque une entrée ou une ressource pour continuer ;
- la *transition* 2 qui intervient lorsque l'ordonnanceur considère qu'il est temps d'attribuer le processeur à un autre processus ;
- la *transition* 3 qui est réalisée lorsque l'ordonnanceur souhaite attribuer le processeur au processus ;
- enfin, la *transition* 4 qui se produit lorsque l'entrée ou la ressource manquante du processus est disponible. Si aucun processus n'est en cours d'exécution, la transition 3 est réalisée, i.e. l'ordonnanceur va attribuer le processeur au processus. Dans le cas contraire, le processus va être en état prêt et donc attendre son tour pour être traité par le processeur.

2.1.2 Les processus légers

Traditionnellement, chaque processus va posséder son propre espace d'adressage et un flot d'exécution unique. Dans certaines situations, il est cependant souhaitable de pouvoir disposer de plusieurs flots d'exécutions qui s'exécutent quasiment en parrallèle et qui se partagent des données dans le même espace d'adressage. Ainsi en considérant les processus comme la conjonction de deux concepts différents : (i) un *moyen de regrouper des ressources* et (ii) un *flot d'exécution*, il est possible d'élaborer un nouveau concept de concurrence qui permet d'exécuter plusieurs flots d'exécution dans un même processus.

Un *flot d'exécution*, nécessairement exécuté dans un processus, va être le regroupement d'un *compteur ordinal* qui effectue le suivi des instructions, de *registres* qui contiennent les variables de travail et d'une *pile* qui contient l'historique d'exécution. Conceptuellement, on peut donc dire qu'un flot d'exécution est comparable à un processus, mis à part que les flots d'exécutions partagent un même espace d'adressage, des fichiers ouverts et d'autres ressources tandis que les processus partagent la mémoire physique, les disques, etc. (voir Fig. 2.2). Puisque les flots d'exécutions partagent un certain nombre de propriétés de processus, ils sont souvent qualifiés de *processus légers*. A l'instar des processus traditionnels, les processus légers vont aussi se retrouver dans les mêmes états et utiliser les mêmes bascules d'un état à un autre (voir Fig. 2.1).

En partageant un même espace d'adressage, les processus légers vont se partager les mêmes variables globales, accéder aux mêmes adresses mémoires, lire, écrire et même effacer des données d'un autre processus léger. En effet, et à la différence des processus, les processus légers appartiennent à la même application qui les a créés dans le but de les faire coopérer étroitement et activement à une même tâche. Cependant, un processus léger va toujours

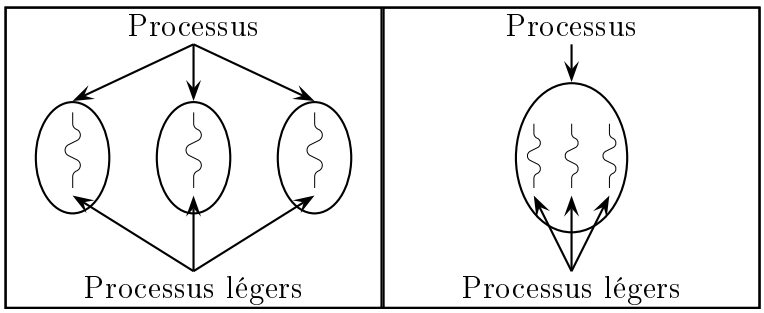


FIG. 2.2 – Différence entre processus et processus légers

posséder sa propre pile d'exécution pour représenter son historique d'exécution. Cette pile d'exécution va être différente entre les processus légers d'un même processus car elle contient les *frames* de chaque procédure invoquée et non terminée. Ce partage du même espace d'adressage pose un certain nombre de problèmes :

- Que se passe-t-il si un processus léger ferme un fichier alors qu'un autre est encore en train de le lire ?
- Supposons maintenant qu'un processus léger a remarqué que la mémoire allait manquer. Il en alloue alors davantage. Un basculement entre processus légers se produit et le nouveau processus léger, ayant fait la même observation, se met lui aussi à allouer de la mémoire...

Bien d'autres complications sont induites par le modèle de processus légers !

En conclusion, concevoir une application avec plusieurs processus légers nécessite une conception soignée et réfléchie qui est indispensable à son bon fonctionnement et qui permet d'éviter de nombreux « comportements erratiques » difficilement reproductibles.

Remarque : Dans la suite de ce manuscrit, je ne ferai plus la distinction entre processus et processus légers et j'utiliserai le terme générique de *processus*. En effet, je considère les deux comme identiques pour les sujets de concurrence et de gestion de la charge qui vont plus particulièrement m'intéresser.

2.1.3 La communication interprocessus

Bien souvent, il est nécessaire de faire coopérer des processus afin de résoudre une tâche donnée en fournissant des mécanismes de *communication inter-processus*. Ces mécanismes doivent prendre en compte les problèmes de passage d'informations entre processus, d'assurer l'absence de conflits lors d'accès concurrents sur une même donnée ou enfin de séquençage des processus, c'est-à-dire faire attendre un processus qui a besoin des résultats d'un autre processus.

Ainsi, le fait que deux processus puissent lire et écrire simultanément des données partagées va bien souvent entraîner des « comportements erratiques » à l'exécution d'une application. Dans le but de résoudre ce problème, il existe une méthode naturelle, appelée

exclusion mutuelle, qui consiste à garantir à un processus accédant à une donnée qu'aucun autre processus ne puisse lui aussi y accèder. La partie du programme à partir de laquelle on accède à ces données partagées est appelée *section critique*.

Mettre en œuvre l'exclusion mutuelle est une question de conception majeure de tout système d'exploitation et quatre propriétés doivent être prises en compte lors de sa conception :

1. deux processus ne peuvent jamais se trouver simultanément dans une même section critique ;
2. aucune supposition ne doit être faite quant à la vitesse et au nombre de processus utilisés par une application ;
3. aucun processus s'exécutant à l'intérieur d'une section critique ne doit bloquer d'autres processus ;
4. enfin, aucun processus ne doit attendre indéfiniment l'accès à une section critique.

Il existe deux grandes techniques de mise en œuvre de l'exclusion mutuelle. La première utilise des attentes actives naturellement consommatrices de temps processeur [92], tandis que la seconde utilise des attentes passives. Dans le cas d'attente passive, les solutions proposées (sémaphores [32] et moniteurs [42]) vont utiliser des mécanismes de mise en sommeil des processus en attente d'accès à une section critique. Lorsque la section critique est « relâchée » l'un des processus en attente va être choisi aléatoirement, *via* un mécanisme de course aux données, par le système pour qu'il puisse accéder à cette section critique. L'ensemble des opérations de vérification, de modification et éventuellement de mise en sommeil vont systématiquement être effectuées dans le cadre d'une *action atomique* unique et indivisible.

Cependant, les sémaphores et moniteurs ont été développés pour résoudre le problème de l'exclusion mutuelle sur un ou plusieurs processeurs ayant tous accès à une mémoire commune. Mais dans un système distribué composé de plusieurs processeurs chacun disposant de sa propre mémoire et connectés en réseau local, ces primitives deviennent inapplicables !

Pour résoudre ce problème, il est nécessaire d'utiliser une autre méthode, l'*échange de messages*. Cette méthode de communication interprocessus va utiliser une procédure pour envoyer un message vers une destination donnée et une seconde pour recevoir un message d'une source donnée. En l'absence de message disponible, le récepteur peut se bloquer jusqu'à ce qu'un nouveau message arrive ou bien retourner immédiatement un code d'erreur. Les systèmes d'échange de messages posent de nombreux problèmes que l'on ne rencontre pas avec les sémaphores ou les moniteurs. En effet, les messages peuvent être perdus par le réseau et pour se prémunir contre ces pertes, un système d'accusé de réception doit être mis en œuvre. Un autre problème est la gestion du nommage et de l'authentification sans ambiguïté des différents processus du système distribué afin d'en garantir la sécurité.

Remarque : Contrairement aux processus « lourds », les processus légers peuvent aisément s'échanger des informations car ils partagent le même espace d'adressage. Cependant, les

problèmes d'exclusion mutuelle et de séquençage sont observables dans le cas des processus légers et les solutions sont aussi applicables aux processus légers.

2.2 Différents types d'E/S

La plupart des applications informatiques réalisent des opérations de lecture ou d'écriture sur un disque dur ou une connexion réseau. Plus particulièrement, les serveurs Internet sont des applications très particulières car la quasi totalité de leurs traitements sont des opérations d'E/S. Actuellement, il existe deux modèles d'E/S différenciés par la nature bloquante ou non bloquante de leur traitement.

2.2.1 Les E/S bloquantes

Dans ce modèle d'E/S, les applications réalisent des E/S qui vont bloquer l'application. Plus précisemment, l'application va bloquer tant que l'E/S n'est pas terminée. Deux cas se présentent :

- soit les données sont transférées ;
- soit il y a une erreur d'E/S.

L'application appelante va ainsi se trouver dans l'état bloqué (voir Fig. 2.1), c'est-à-dire qu'elle ne va pas consommer de temps processeur et va attendre d'être réveillée par l'ordonnanceur quand les données sont disponibles sur l'interface d'E/S, i.e. le disque dur ou la connexion réseau (voir Fig. 2.3).

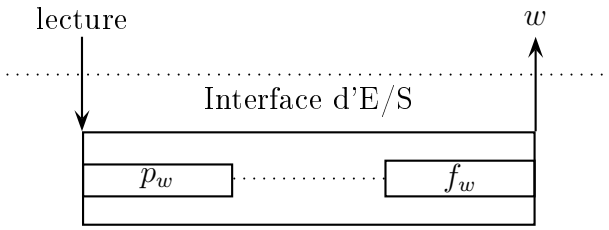


FIG. 2.3 – Les E/S bloquantes

Le comportement de ce modèle d'E/S est bien connu, car il est certainement le modèle le plus couramment utilisé dans les systèmes d'exploitation actuels. En effet, lorsque l'application réalise un appel système d'E/S, elle va bloquer et basculer en mode noyau. L'E/S est alors initiée sur une interface donnée. Une fois les données physiquement lues, elles sont déplacées dans le tampon de données de l'espace utilisateur, l'application est réveillée et peut continuer à s'exécuter.

2.2.2 Les E/S non bloquantes

Dans ce modèle d'E/S, les applications sont capables de « chevaucher » le traitement des différentes E/S. Lors d'un appel à une procédure d'E/S, celui-ci va immédiatement retourner en indiquant que l'opération a été correctement initiée. L'application va pouvoir

réaliser d'autres traitements tandis que l'opération d'E/S se poursuit en « arrière-plan ». Dès que des données sont disponibles sur l'interface, un signal est généré signalant à l'application qu'elle peut lire des données sur l'interface en question (voir Fig. 2.4).

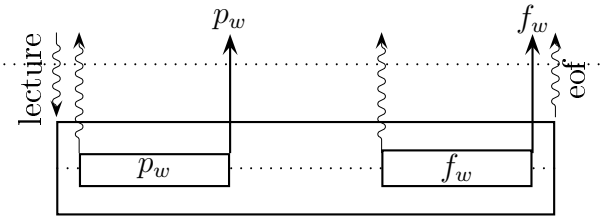


FIG. 2.4 – Les E/S non bloquantes

La capacité de chevaucher les traitements et les E/S dans un même processus exploite la différence de vitesse de traitement du processus et la vitesse de traitement des E/S beaucoup plus lentes. Le processeur peut ainsi réaliser d'autres tâches, ou dans le cas de serveurs Internet, traiter les E/S qui sont déjà terminées et en initier de nouvelles. Ce modèle d'E/S est donc extrêmement intéressant et performant dans le cas de serveurs Internet !

Il est nécessaire de mettre en place des mécanismes pour récupérer le signal indiquant que des données sont disponibles sur une interface. Ce *mécanisme de multiplexage* des E/S est mis en place à l'aide d'une procédure telle que *poll* sur System V ou *select* sur BSD Unix. La sélection va surveiller l'activité de toutes les interfaces d'E/S (fichiers ou connexions réseaux). Si une activité se produit pour une interface d'E/S alors l'application va lire ou écrire les données ou une partie de celles-ci pour cette interface d'E/S.

2.3 Modèles de concurrence

Les serveurs Internet doivent faire face à une concurrence massive en répondant aux différents clients en un minimum de temps et être robuste face à des montées en charge très brusques et très variables. De plus, ils vont consommés plus ou moins de ressources car un serveur Internet peut être utilisé pour configurer un pda ou répondre aux requêtes de milliers de clients sur une machine multi-processeurs. Pour répondre à ces problématiques, il existe actuellement plusieurs techniques de multiplexage de différentes activités au sein d'une même application, appelées *modèles de concurrence*, qui vont être plus ou moins efficaces selon l'architecture matérielle sous-jacente.

Fournir une classification de ces modèles de concurrence pour en faciliter l'étude n'est pas une nouvelle idée [1, 86, 114]. La classification présentée ici (voir Fig. 2.5) est néanmoins beaucoup plus simple et se base sur deux critères que je considère comme prépondérants dans la spécification d'un modèle de concurrence [69] :

- le type d'E/S (bloquantes ou non bloquantes) ;
- l'utilisation des processus (un seul processus, coopération ou compétition).

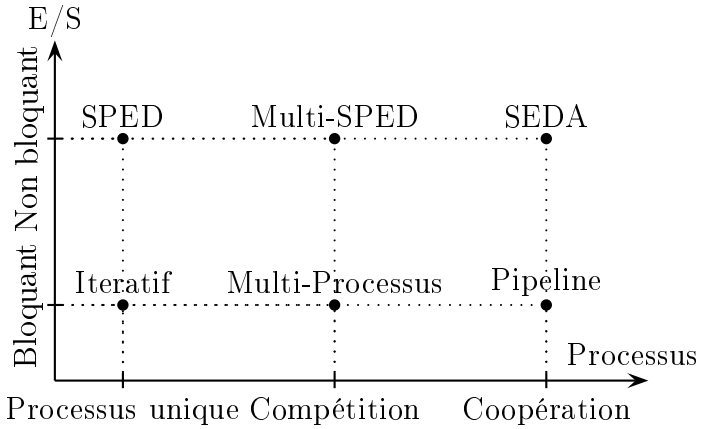


FIG. 2.5 – Taxonomie des modèles de concurrence

La *compétition* entre processus apparaît lorsque plusieurs processus réalisent le même code et concourent pour le traitement des données. Inversement, la *coopération* se produit lorsque plusieurs processus coopérent en vue de réaliser une tâche donnée. Ainsi, ces processus vont réaliser un travail différent, communiquer en se partageant des ressources et se synchroniser pour éviter des interblocages.

L'utilisation de ces deux critères permet de s'abstraire de toutes les variantes implantatoires ou optimisantes de chacun de ces modèles telle que l'utilisation de vivier de processus ou l'absence de support des E/S non bloquantes pour les disques durs de certains systèmes d'exploitation [86].

2.3.1 Architecture itérative

L'architecture itérative utilise un seul processus et des E/S bloquantes pour traiter plusieurs requêtes (voir Fig. 2.6). Ainsi avant d'accepter une nouvelle requête, cette architecture va finir l'ensemble des traitements requis par la requête précédente. Elle utilise un tampon de requêtes dédié qui respecte la politique de gestion de file, i.e. *premier arrivé, premier servi*.

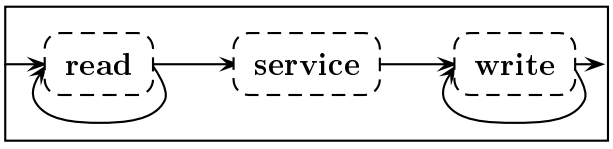


FIG. 2.6 – L'architecture itérative

Cette architecture ne présente pas de comportement concurrent car il n'y pas d'entrelacement du traitement de plusieurs requêtes. Néanmoins, elle est vraiment très simple à programmer et peut être utilisée dans le cas de serveur utilisé sporadiquement.

2.3.2 L'architecture Single-Process Event-Driven (SPED)

L'architecture SPED [86] utilise un seul processus pour traiter plusieurs requêtes et va utiliser des E/S non bloquantes (voir Fig. 2.7).

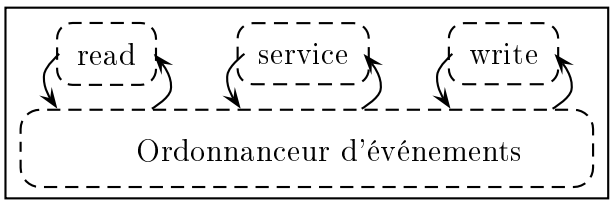


FIG. 2.7 – L'architecture SPED

Dans cette architecture, une application est représentée comme une *machine à états finie*. Le *traitement* d'une requête va correspondre au chemin suivi dans la machine à états finie par la requête. La requête sera servie lorsqu'un état de fin valide est atteint, i.e. soit une erreur soit une réponse. Un *état* va correspondre à une séquence d'instructions et être associé à un événement d'E/S particulier, tel qu'un événement de lecture ou d'écriture. Lorsqu'un événement d'E/S se produit, il est sélectionné et l'état associé à cet événement va être exécuté pour la requête en cours. Grâce à l'utilisation d'E/S non bloquantes, l'architecture SPED va entrelacer le traitement de plusieurs requêtes.

Cependant, cette architecture est difficile à implanter et beaucoup de systèmes d'exploitation ou d'implantations d'E/S non bloquantes ne fournissent pas le support dans le cas des accès disques [86]. C'est pourquoi, la variante AMPED (*Asymmetric Multi-Process Event-Driven*) a été proposée. Cette architecture va utiliser un vivier de processus d'E/S qui permet de simuler le comportement non bloquant des accès disques.

2.3.3 L'architecture multi-processus

L'architecture multi-processus [62] utilise plusieurs processus exécutant les mêmes traitements et des E/S bloquantes (voir Fig. 2.8).

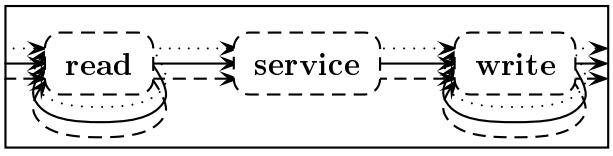


FIG. 2.8 – L'architecture multi-processus

Dans cette architecture, chaque requête cliente va être attribuée à un processus réalisant l'ensemble des traitements du serveur. Comme plusieurs processus peuvent être utilisés et ordonnancés sur les architectures modernes, ce modèle de concurrence va pouvoir traiter plusieurs requêtes clientes « en même temps ». La gestion des traitements du disque, des connexions réseaux et du processeur de chaque processus est réalisée par le système d'exploitation et ce, de façon transparente pour l'utilisateur.

Il existe plusieurs méthodes de gestion des processus dont la plus naturelle est la création d'un nouveau processus pour chaque requête cliente. Cependant, le temps de création d'un processus (ou d'un processus léger) est coûteux et il est peu performant d'adopter cette approche. Une autre méthode est d'utiliser un vivier de processus qui permettra d'amortir le temps de création des processus au démarrage du serveur en empruntant et relâchant « à la demande » des processus de ce vivier lors de l'exécution du serveur. Cependant, cette approche a l'inconvénient d'utiliser en permanence des ressources (stockage en mémoire du vivier de processus) même si le serveur n'a aucune activité.

2.3.4 L'architecture multi-SPED

L'architecture multi-SPED ou symétrique SPED [69, 88] est une extension naturelle du modèle SPED. Cette architecture utilise plusieurs processus et des E/S non bloquantes (voir Fig. 2.9). Contrairement à l'approche N-copie [116] qui utilise une même adresse mais des ports différents pour chaque processus, l'architecture mSPED va utiliser la même adresse et le même port pour tous les processus.

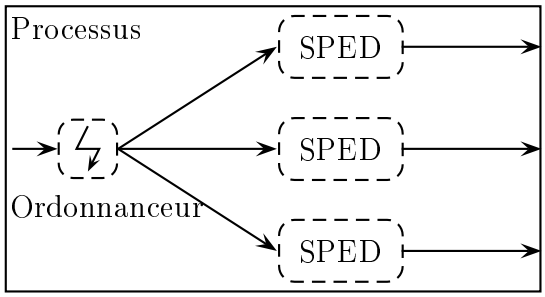


FIG. 2.9 – L'architecture multi-SPED

Dans cette architecture, chaque processus est un serveur pleinement fonctionnel de type SPED. Cependant lorsque l'un des processus bloque, par exemple sur une E/S disque, cette architecture a la possibilité de changer de processus courant et d'en exécuter un autre. Ainsi sur une architecture mono-processeur, lorsqu'un serveur Internet de type SPED bloque, le modèle mSPED va pouvoir exécuter un autre processus et ainsi utiliser pleinement le processeur.

Différentes techniques, similaires aux méthodes d'ordonnancement des architectures multi-processeurs (tourniquet, prime, etc.), peuvent être utilisées afin d'attribuer une requête cliente à un processus de type SPED.

2.3.5 L'architecture pipeline

L'architecture pipeline utilise plusieurs processus coopérants et des E/S bloquantes afin de traiter plusieurs requêtes en même temps [62, 109, 114]. Similairement à SPED, cette architecture exploite la modélisation sous forme de machine à états finie des différentes étapes nécessaires au traitement d'une requête (voir Fig. 2.10).

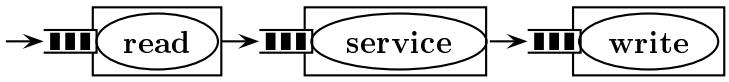


FIG. 2.10 – L'architecture pipeline

Un état de cette machine va correspondre à une unité fondamentale de traitement exécutée dans un processus. Les différents processus vont s'échanger des informations via des files locales. Un processus va ainsi traiter chacun des événements présents dans sa file d'entrée et produire de nouveaux événements représentant le(s) résultat(s) de son exécution qu'il placera dans les files de ses successeurs. Cette architecture se comporte exactement comme le pipeline présent dans la microarchitecure d'un processeur [3].

L'un des problèmes majeurs de cette architecture est le contrôle automatique de la surcharge du serveur, par rejet d'événements en entrée de chaque état car les files sont bornées. En effet, chaque état va prendre des décisions locales et indépendantes des autres. Cette automatisation peut entraîner des difficultés lors du développement des couches applicatives de protocoles de communication car un client peut continuer à dialoguer avec un serveur ayant rejeté certains de ses événements. Un second problème vient du non respect de l'ordre des événements qui augmente la complexité du développement de l'application.

2.3.6 L'architecture Staged Event-Driven (SEDA)

Cette architecture, du nom d'une de ses implantations [109], utilise plusieurs processus coopérants et des E/S non-bloquantes [62, 109, 114]. Contrairement à l'architecture pipeline, chaque processus va être capable de traiter plusieurs requêtes clientes en même temps (voir Fig. 2.10).

Remarque : Il existe différentes variantes aux deux architectures présentées ci-dessus, l'une des plus naturelles est sans aucun doute d'utiliser plusieurs processus pour réaliser un même traitement fondamental [109, 114].

2.4 En conclusion...

Actuellement, il existe plusieurs techniques pour mettre en œuvre la concurrence au sein des applications. J'ai caractérisé ces techniques selon (i) l'utilisation des processus et (ii) le type d'E/S utilisée (bloquante ou non bloquante). Ainsi, il existe deux utilisations majeures des processus. On parle de *compétition* entre processus lorsque l'ensemble des processus réalisent les mêmes traitements et concourrent pour traiter les données entrantes. Inversement, on parle de *coopération* entre processus lorsqu'ils réalisent des tâches distincts et s'échangent des informations en vue de traiter les données entrantes. Dans le cas des E/S, le mode non bloquant nécessite de gérer la sauvegarde manuelle du contexte de lecture. Ce qui n'est pas nécessaire dans le cas d'E/S bloquante car, on est assuré de récupérer l'ensemble des données à lire lorsque les instructions de lecture sont terminées. Pratiquement, les architectures non bloquantes sont très sensibles au nombre maximum de descripteurs utilisés en même temps

sur le serveur (voir 6.3). Chacun des modèles de concurrence va présenter des avantages et des inconvénients résumés par le tableau suivant :

Modèle de concurrence	Avantages	Inconvénients
Itératif	Coût faible en ressources Coût faible développement	Pas de concurrence
SPED	Coût faible en ressources Performant	Très dur à développer Limite physique (nombre de descripteurs)
Multi-processus légers	Facile à développer Performant	Risque d'interblocages Limite physique (nombre de processus légers)
MSPED	Très performant Bonne gestion de la charge	Extrêmement dur à développer Risque d'interblocages
Pipeline	Bonne gestion de la charge Application distribuée naturellement	Risque d'interblocages Nombre de clients en concurrence borné par la taille du pipeline E/S bloquante
SEDA	Bonne gestion de la charge Application distribuée naturellement	Risque d'interblocages Limite physique (nombre de descripteurs) Dur à développer

Lors du développement d'une application, le modèle de concurrence va (i) fortement structurer le code de celle-ci et (ii) consommer plus ou moins de ressources systèmes. Ainsi, passer d'un modèle à un autre va bien souvent nécessiter le redéveloppement de « bout en bout » de l'application pour un nouveau modèle de concurrence. Il y a donc un véritable manque de rationalisation en temps et en coût de développement des applications concurrentes. De plus, pour chacune des architectures matérielles existantes, il existe un modèle de concurrence qui permet d'optimiser les deux critères que sont : (i) les performances de l'application et (ii) la consommation de ressources.

Est-il possible de fournir une méthode de développement accompagnée d'outils pour passer simplement et rapidement d'un modèle de concurrence à un autre sans modifier le code « métier » d'une application ?

3.

Saburo, un outil de développement de serveurs Internet

Sommaire

Actuellement, la qualité d'un logiciel est une préoccupation prépondérante lors de son développement. Ainsi, son absence de « comportements non désirés », son évolution vers de nouvelles fonctionnalités, sa souplesse d'utilisation ou son exécution sur une très grande variété d'architectures matérielles de façon répartie ou distribuée sont des critères importants à tout bon logiciel. De plus, la pression du marché impose des temps de développement plus courts et des coûts plus faibles aux industriels.

L'informatique est ainsi passée d'une ère « artisanale » à une véritable ère « industrielle » avec d'importants soucis de rationalisation !

Ces évolutions ont et vont continuer à bouleverser la manière de concevoir un logiciel. C'est pourquoi, il devient de plus en plus difficile de développer une solution logicielle de « bout en bout » sans avoir à utiliser des composants fournis « sur étagères » par d'autres partenaires. De plus, pour faciliter la communication entre les différents acteurs, les technologies propriétaires disparaissent progressivement et laissent place à des technologies standardisées par des consortiums, le W3C (*World Wide Web Consortium*) [112] ou l'OMG (*Object Management Group*) [83]. Ces nouvelles exigences vis-à-vis des logiciels proviennent de contraintes liées au temps de développement, à son financement, aux disparités de connaissances des différents utilisateurs ainsi qu'aux nécessités d'adaptations rapides aux besoins du marché. Enfin, les logiciels doivent aussi offrir une *convivialité d'utilisation et de configuration* par le biais d'interfaces utilisateur interactives, une *facilité d'utilisation* qui nécessite peu de compétences informatiques, une *adaptabilité rapide à leur environnement* qui s'appuie

sur une implantation flexible et modulaire et une *communication aisée* entre ses composants ou avec des applications externes *via* un format d'échange standardisé.

Je pense que ces nouvelles contraintes doivent être prises en compte lors du développement d'un serveur Internet. Cependant, la plupart des développeurs de serveurs se focalisent sur l'optimisation des performances [12, 86, 88, 114]. Ainsi du fait des nombreuses optimisations de code, de l'imbrication de la partie « métier » et des différentes préoccupations, dont la concurrence, le code source des serveurs Internet est difficilement lisible, maintenable et évolutif. Ce qui va « à contre courant » des nouvelles préoccupations du processus de développement !

D'un point de vue conception, le modèle de développement abordé dans Saburo est très similaire à l'approche choisie dans Darwin/Regis [73]. Ainsi, Darwin est un langage de configuration utilisé pour définir des compositions hiérarchiques de composants interconnectés. Les abstractions principales gérées par Darwin sont des composants et des services qui spécifient les interactions entre composants. Darwin est complété par Regis [72], un environnement de programmation qui permet le développement et l'exécution de programmes distribués. De même que Saburo, la structure des programmes est séparée entre la partie communication et la partie calcul. Ainsi, les éléments sont combinés pour former les programmes distribués. Contrairement à Saburo, Darwin n'a pas été conçu pour la vérification automatique et n'abstrait pas la partie concurrence d'une application.

Dans ce chapitre, je vais présenter mon modèle de développement de serveurs Internet qui est basé sur la notion de *fabrique de logiciels* [89]. Du fait de l'intégration à différents niveaux des concepts et techniques qui tentent de répondre aux nouvelles contraintes de développement, je vais commencer par les décrire succinctement. Puis, je vais présenter Saburo, un prototype en Java de mon modèle de développement. Enfin pour illustrer l'utilisation de Saburo, je vais détailler le développement d'un serveur HTTP simple.

3.1 Un modèle de développement

La dernière décennie a vu se poursuivre l'accroissement exponentiel de la complexité des logiciels informatiques. Pour réduire cette complexité, de nouvelles techniques sont apparues visant à minimiser simultanément les coûts de développement et le temps de mise sur le marché des nouveaux logiciels. L'une des solutions les plus simples est d'utiliser la *modélisation* pour essayer de maîtriser cette complexité, tant pour produire le logiciel que pour le valider. Au sens large du terme, la *modélisation* est l'utilisation d'une représentation simplifiée d'un aspect de la réalité pour un objectif donné. En informatique, la modélisation peut être vue comme la séparation des différents besoins fonctionnels et préoccupations extra-fonctionnelles (sécurité, fiabilité, efficacité, performance, flexibilité, etc.) issus des exigences du logiciel. La *conception* du logiciel consiste alors à *fusionner* ou *tisser* des solutions à ces différentes préoccupations avec le code « métier ».

Ce processus n'est en aucun cas nouveau et bien que l'on utilise de nouveaux noms,

ces activités d'abstraction - modélisation et conception - tissage existent depuis toujours. Cependant dans la plupart des cas, les modèles et les solutions de conceptions restent implicites, ou tout au moins informels et sont appliqués manuellement. Dans cette section, je vais présenter un modèle de développement qui, à partir d'une description indépendante du modèle de concurrence, permet de *produire automatiquement* un serveur Internet pleinement fonctionnel.

3.1.1 Préliminaires

L'application de nouveaux concepts (encapsulation, polymorphisme, héritage ou masquage d'informations) qui sont propices à la modularité, la réutilisabilité et l'extensibilité du code source a été grandement facilitée par les *langages de programmation par objets*. Ces langages ont même tellement bien répondu aux différentes attentes des développeurs qu'il est difficilement envisageable actuellement d'obtenir une modularité, une réutilisabilité et une extensibilité du code source sans les utiliser.

Cependant, ce style de programmation a principalement introduit une relation « verticale » entre les classes et apparaît de plus en plus insuffisant pour prendre en compte des préoccupations qui seraient « transversales » entre celles-ci. C'est pourquoi, les principales avancées en architecture et ingénierie logicielle concernent maintenant la définition de nouveaux outils, par exemple la *programmation par aspects* [17] ou la *programmation par composants* [76], pour pallier au problème de transversalité des préoccupations et systématiser l'application de l'un des plus importants concepts du développement logiciel : la *séparation des préoccupations* [50].

3.1.1.1 La séparation des préoccupations

Le principe de séparation des préoccupations est de répondre, pour l'essentiel, aux problèmes de mise en œuvre simple et rapide de nouvelles fonctionnalités (sécurité, répartition, temps réel, etc.) au sein des applications.

Définition 3.1 *Séparation des préoccupations*
Les parties d'un système qui correspondent à des domaines et des responsabilités différents doivent être spécifiés séparément et composables automatiquement.

Une préoccupation va correspondre systématiquement à un domaine d'expertise précis et présenter des responsabilités bien identifiées dans un logiciel. Les besoins en terme de répartition, de dynamicité, de composition des logiciels sont de plus en plus conséquents et l'implantation des différentes préoccupations devient une opération de plus en plus complexe, ce qui nécessite l'appel à des experts. Pour permettre la focalisation sur une unique préoccupation de ces experts et une plus grande réutilisation du travail effectué, il est nécessaire de pouvoir les spécifier et les implanter séparément. La conception d'un logiciel est alors plus facilement maîtrisée car l'impact des évolutions d'une partie d'un logiciel sur les autres est ainsi fortement limité.

Cependant l'introduction de ce lien faible entre les différentes préoccupations et le code « métier » d'un logiciel nécessite un mécanisme, dépendant de la technique de séparation, qui permet de faire la composition entre les différentes parties d'un système en vue de le rendre cohérent [50].

Enfin, pour appliquer correctement ce principe de séparation des préoccupations et donc proposer des outils permettant de systématiser et faciliter son application, trois questions primordiales doivent être soulevées :

1. Comment identifier les différentes préoccupations présentes au sein d'une application ?
2. De quelle manière peut-on spécifier séparément les différentes préoccupations ainsi détectées ?
3. Enfin, comment les composer en vue d'obtenir un système complet ?

3.1.1.2 La programmation par aspects

L'une des techniques permettant de mettre en œuvre le principe de séparation des préoccupations est la *programmation par aspects* [17] dont l'implantation la plus connue est AspectJ [9]. L'idée de la programmation par aspects est de gérer, de manière modulaire, les préoccupations en les séparant du code de base. C'est une technique dite *non invasive*.

En effet, les aspects introduisent un découpage transversal et la composition repose sur des points d'entrelacement des portions de code source. Parmi les aspects souvent traités nous trouvons la gestion de l'authentification, l'archivage des données (persistance), l'application de patrons de conception, etc.

Rendre un point duplicable avec AspectJ

La classe Point définit un point géométrique en coordonnées rectangulaires ainsi que des opérations simples pour déplacer un point. L'implantation proposée de la classe Point est minimale. Elle est intégralement dédiée à la gestion d'un point. C'est-à-dire qu'il n'y a aucune méthode pour comparer, dupliquer, etc. un point.

```
public class Point {
  protected double x = 0;
  protected double y = 0;

  public double getX() {
    return x;
  }

  public double getY() {
    return y;
  }

  public void set(double newX, double newY) {
    x = newX;
    y = newY;
```

```
  }
  public void offset(double deltaX, double deltaY) {
    x = x + deltaX;
    y = y + deltaY;
  }
}
```

L'aspect que je souhaite implanter va réaliser la duplication d'un point ce qui correspond à l'implantation de l'interface Cloneable en Java. L'aspect va déclarer qu'un Point implante l'interface Cloneable.

```
public aspect CloneablePoint {
  declare parents : Point implements Cloneable;
  ...
}
```

Il spécifie aussi l'implantation de la méthode clone() qui implante effectivement cette duplication.

```
public aspect CloneablePoint {
  declare parents : Point implements Cloneable;
  public Object Point.clone() throws CloneNotSupportedException {
    return super.clone();
  }
}
```

En Java, toutes les classes héritent par défaut de la méthode clone de la classe Object mais, un objet n'est réellement duplicable que s'il implante l'interface Cloneable. Fréquement, les objets nécessitent plus qu'une simple copie physique (copie bit à bit) et le développeur aura la charge d'implanter correctement cette duplication. La méthode clone de la classe Object ne réalise qu'une copie physique d'un objet !

Sur l'exemple présenté ci-dessus, il apparait très clairement qu'un aspect est ajouté *a posteriori* à une classe et ce, de manière non invasive. Il est donc très facile, car sans modification directe, d'étendre le code « métier » d'un logiciel par simple ajout de nouvelles fonctionnalités. Néanmoins, cette technique soulève quelques questions :

1. Où tisser les aspects ?
2. Comment composer plusieurs aspects ?
3. Quelle technique d'implantation choisir ?

Remarque : Cette dernière question peut être résolue en utilisant la transformation de programme, la réflexivité ou la génération de code adapté.

3.1.1.3 La programmation par composants

Une seconde limitation à l'approche par objets est sa granularité très fine qui est peu adaptée aux systèmes complexes. Pour répondre à ce problème, un nouveau concept a été introduit : le *composant* [76, 106]. Un composant va encapsuler plusieurs objets pour proposer un service « métier » et lui associer, le plus facilement possible, du code non-fonctionnel (persistance ou une politique de sécurité). De plus cette approche permet de déployer et répartir facilement une application. La communication entre composants se fait alors par des connexions réseaux. Enfin, il est possible de construire une application complète par simple assemblage de composants fournis « sur étagères » ce qui permet de réduire les coûts et les temps de développement.

Il existe actuellement quatre principaux modèles de composants :
- les EJBs (*Enterprise Java Beans*) de Sun [101, 105] ;
- CCM (*CORBA Component Model*) de l'OMG [108] ;
- l'association DCOM (*Distributed Component Object Model*) et .NET de Microsoft [78] ;
- enfin, les *web services* du W3C [113].

Bien que la création d'applications par simple assemblage de composants attire de nombreux industriels, il reste encore de nombreuses interrogations :

1. Comment faire communiquer des composants issus de modèles différents ?
2. Comment découvrir les composants ayant les services souhaités ?
3. Comment rendre adaptable les composants ?

3.1.1.4 L'ingénierie dirigée par un modèle

Les nouvelles contraintes imposées aux logiciels ont aussi été prises en compte lors de leur spécification avec l'apparition de nouvelles méthodes d'analyse et de conception. Ces méthodes, dont la plus célèbre est UML (*Unified Modelling Language*) [84] de l'OMG, vont utiliser des modèles pour spécifier, assembler et visualiser un logiciel complexe mais aussi, pour produire automatiquement la trame de son code source.

Plus récemment, l'OMG a introduit une nouvelle approche d'écriture de spécifications et de développement d'application, nommée MDA (*Model-Driven Architecture*) [15]. L'idée centrale de l'approche MDA est l'élaboration de modèles indépendants de toute plate-forme ou technologie, nommés PIM (*Platform Independent Model*), qui sont ensuite transformés vers un ou plusieurs modèles dépendants de plates-formes spécifiques, nommés PSM (*Platform Specific Model*). Le passage d'un PIM vers un ou plusieurs PSMs va s'effectuer *via* des règles de transformation implantées à l'aide de générateurs (voir Fig. 3.1 et section 3.1.1.5 sur la programmation par génération).

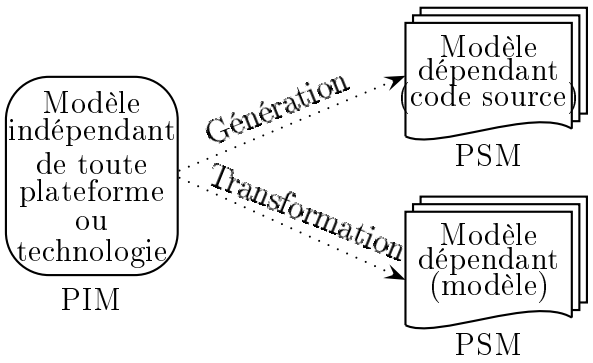


FIG. 3.1 – L'approche *Model-Driven Architecture*

L'approche MDA permet une séparation claire entre : (i) la partie « métier » d'un logiciel et (ii) la partie technologie cible. Elle offre aussi une plus grande pérennité et portabilité de la partie « métier » car celle-ci est complètement indépendante de toute technologie. Enfin, elle permet de faire évoluer très simplement une application en modifiant seulement les générateurs.

3.1.1.5 La programmation par « génération »

L'utilisation des logiciels a permis d'automatiser un certain nombre de tâches dont le développement de logiciels. L'automatisation de la production de logiciels permet non seulement, une réduction du temps et des coûts de développement mais aussi, l'augmentation de la qualité des logiciels et la suppression d'un certain nombre d'erreurs. De plus, les coûts et la difficulté de maintenance des logiciels sont aussi réduits grâce à cette automatisation. Cependant, la génération automatique et intégrale d'un système applicatif ne peut être réalisée facilement et cela semble peu probable dans un futur proche.

Problématique

Ces considérations ne sont pas nouvelles car la programmation par objet nécessite un plus haut degré d'abstraction mais, elle ne permet pas une automatisation du développement. Les patrons de conception [37] sont devenus des standards *de facto* pour un très grand nombre de développeurs mais, ils sont utilisés *a posteriori*, i.e. lors de la phase de développement, et leur mise en œuvre est laissée à la charge du développeur. L'idée fondamentale de la programmation automatisée est donnée par la citation suivante [29] :

« Si on peut composer manuellement les composants, il est possible d'automatiser ce processus. »

Le terme *programmation générative* est survenue pour la première fois en 1996 et a fait l'objet du livre [30].

Principe

La programmation générative est un nouveau paradigme du développement de logiciel. Elle ne rivalise ni avec les paradigmes présentés auparavant ni avec la programmation orientée objet mais, les complète. En plus de l'automatisation du développement de logiciels, le principe fondamental de la programmation générative est la prise de conscience de l'existence de *familles de logiciels*. En effet, le développement conventionnel nécessite de développer des composants ou des systèmes entièrement même s'ils présentent des caractéristiques semblables. Au contraire, les principes de programmation générative supposent qu'il est possible de produire automatiquement les membres d'une même famille. Cette génération est réalisée sur la base d'un modèle commun nommé *modèle commun de génération* qui est constitué de trois éléments :

1. une méthode pour spécifier les membres d'une famille ;
2. des modules qui permettent la création de chaque membre ;
3. le savoir-faire de configuration pour transcrire les spécifications en implantation.

La *modélisation* et la *spécification* des familles sont réalisées à l'aide de techniques comme l'*ingénierie de domaine*, la *modélisation des comportements* et les *langages spécifiques à un domaine* (*Domain Specific Language*).

La *conception des modules* et l'*implémentation des configurations*, c'est-à-dire la transcription des spécifications en logiciels et composants exigent des *outils de générations* adéquats.

Outils de génération

La tâche principale d'un outil de génération ou *générateur* est de traduire la spécification de haut niveau dans un format de niveau plus bas. Les compilateurs sont l'exemple le plus connu de générateur. Il existe différents types de générateur :

- les *générateurs spécialisés* sont dédiés à des tâches très particulières (générateurs d'interface graphique utilisateur, transformateurs de spécification UML, pré-compilateurs DBMS – *DataBase Management System*). Ces générateurs vont travailler efficacement dans leur domaine de responsabilité mais du fait de leur spécialisation, ils ne sont pas flexibles et ne peuvent pas être adaptés à d'autres tâches ;
- les *générateurs « à faire soi même »* ont une flexibilité importante car il faut spécifier des macros ou patrons dans les compilateurs eux-mêmes. Cependant, ces générateurs n'ont pas de réelle « intelligence » individuelle. Il y a donc un gros coût de développement à l'usage ;
- les *combinaisons des deux principes précédents* consistent en une partie prédéterminée et spécialisée pour des fonctions de base et ils peuvent être étendus par l'utilisateur à l'aide de macros ou de patrons.

3.1.2 Proposition de « patrons de conception » de concurrence

Les investigations initiales sur les modèles de concurrence présentées dans le chapitre 2 ont montré qu'il était nécessaire de proposer des méthodes systématiques afin de faciliter le développement des applications concurrentes et plus précisémment les serveurs Internet. C'est pourquoi, je vais proposer dans cette section un « patron de conception de concurrence » pour chaque modèle présenté dans la section 2.3.

Un *patron de conception* est défini comme la spécification d'une solution récurrente à un problème standard [37]. Les patrons de conception définis ici seront, par la suite, utilisés dans un modèle de développement permettant de générer automatiquement des serveurs Internet à partir d'une spécification indépendante de tout modèle de concurrence 3.1.3.

3.1.2.1 Définitions préliminaires

Je vais définir par *service* la suite d'instructions qu'un serveur Internet doit réaliser pour répondre à une requête émise par un client. Un service va être constitué d'une suite d'opérations de haut niveau, que je vais appeller *stages* en référence à l'architecture SEDA [109]. Les stages peuvent être des instructions de calcul, des opérations d'E/S, etc.

Par exemple pour répondre aux requêtes de ses clients, un serveur HTTP (voir Fig. 3.2) va devoir *accepter et établir* une nouvelle connexion réseau avec chaque client. Puis, après avoir *lu* et *décoder* la requête entrante, il va chercher à *servir* cette requête. Il va ensuite constituer sa réponse, l'*encoder* et finalement l'*envoyer* au client.

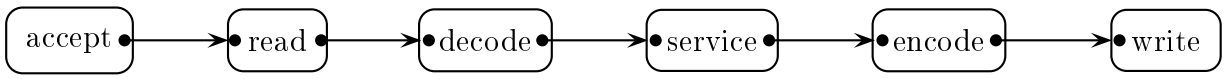


FIG. 3.2 – Le service HTTP et ses différents stages

L'intérêt de découper un service en plusieurs stages est de distinguer très facilement :

1. le code fonctionnel du service ;
2. le code lié au modèle de concurrence utilisé.

En effet, les modèles de concurrence (voir section 2.3) sont caractérisés par des technologies différentes (le type des E/S et l'utilisation des processus). Ainsi lors du développement d'un serveur Internet, le code fonctionnel ne va jamais changer mais, la structure et l'organisation de l'implantation de chacun des modèles de concurrence vont être radicalement différentes d'un modèle à un autre !

La structure du code va ainsi correspondre au patron de conception associé au modèle de concurrence. Les modèles de concurrence vont s'exprimer selon la méthode d'encapsulation des stages dans les processus légers : (i) *tous les stages s'exécutent-ils dans un seul processus ?* (ii) *Dans plusieurs processus ?* (iii) *Coopérent-ils ou concourrent-ils ?*, leur méthode

d'interconnection : (i) files locales, (ii) appels de fonction ou (iii) connexions réseau et enfin, selon le type des E/S : (i) bloquantes ou (ii) non bloquantes.

Je vais maintenant détailler les différents patrons de conception des modèles de concurrence pris en compte par mon approche.

3.1.2.2 Architecture itérative

Cette architecture utilise un seul processus léger dans lequel les stages sont exécutés séquentiellement. Les stages vont être connectés entre eux *via* des appels de méthodes et les E/S utilisées dans cette architecture sont bloquantes.

3.1.2.3 Architecture SPED

Les différents stages d'un service sont exécutés dans un seul processus léger et sont connectés *via* des appels de méthode. Contrairement à l'architecture précédente, les E/S utilisées sont non bloquantes. Il est donc nécessaire de mettre en place un mécanisme permettant d'associer à un événement d'E/S, un stage donné. De plus, les stages peuvent être issus d'un découpage logique d'un service. Un événement donné peut donc être associé à un sous-graphe, i.e. une succession de stages.

3.1.2.4 Architecture multi-processus légers

Cette architecture utilise plusieurs processus légers dans lesquels les stages vont être exécutés séquentiellement. L'ensemble de ces processus va réaliser exactement les mêmes traitements ! Cependant un mécanisme doit être fourni pour attribuer une tâche à un processus de traitement. Les stages vont être connectés entre eux *via* des appels de méthodes et les E/S utilisées dans cette architecture sont bloquantes.

3.1.2.5 Architecture multi-SPED

Cette architecture est composée de plusieurs processus légers, chacun utilisant l'architecture SPED. L'ensemble des processus vont réaliser exactement les mêmes traitements. Similairement à l'architecture multi-processus, un processus dédié va accepter un nouveau client et transmettre à un processus de traitement la requête cliente. Le mécanisme de transmission peut s'inspirer de ceux utilisés pour l'ordonnancement des tâches d'un processeur.

3.1.2.6 Architecture Pipeline

Cette architecture utilise plusieurs processus légers, chacun ayant en charge d'exécuter un stage. Afin de communiquer entre eux, les stages vont utiliser des files locales pour s'échanger des informations. Les E/S utilisées dans cette architecture sont bloquantes.

3.1.2.7 Architecture SEDA

Les différents stages d'un service vont être exécutés dans un processus légers propre. Dans le but de communiquer entre eux, les stages vont utiliser des files locales. Les E/S sont synchrones ce qui implique la mise en place d'un mécanisme permettant d'associer à un événement d'E/S le stage à exécuter.

3.1.3 Description de mon modèle de développement de serveurs Internet

La prise en compte des nouvelles approches de développement nécessite un changement radical dans la façon d'appréhender la conception et le développement d'applications. C'est pourquoi, je me suis focalisé sur la proposition d'une nouvelle méthodologie de développement de serveurs Internet qui intégre à différents niveaux des approches similaires à (i) l'ingénierie dirigée par un modèle (MDA), (ii) la programmation par aspects et (iii) la programmation par composants.

3.1.3.1 Problématique

Du fait de nombreuses optimisations de code, de l'imbrication de la partie « métier » et des différentes préoccupations, dont la concurrence, le code source des serveurs Internet est difficilement lisible, maintenable et évolutif. De plus, les modèles de concurrence vont avoir des coûts plus ou moins importants en ressources. Ainsi, un serveur Internet peut être utilisé pour configurer un pda *via* une connexion réseau ou tenter de répondre aux requêtes de milliers de clients sur une machine multi-processeurs. Les ressources disponibles vont alors être extrêmement diverses et le serveur Internet devra s'adapter aux besoins du développeur et aux ressources disponibles. Afin de s'adapter à l'architecture matérielle sous-jacente et vue l'absence de consensus sur le meilleur modèle de concurrence [11, 41, 85], il est nécessaire de redévelopper de « bout en bout » le même serveur Internet pour un autre modèle de concurrence. C'est pourquoi, je pense qu'il est nécessaire de proposer une méthode permettant de passer très facilement et de manière transparente d'un modèle de concurrence à un autre.

Le choix du modèle de spécification dans une approche d'ingénierie dirigée par un modèle est déterminant !

En effet, il doit être *simple* pour faciliter son utilisation et *suffisament expressif* pour décrire des propriétés évoluées d'un système complexe. De plus, je pense qu'*utiliser des langages existants* en proposant des bibliothèques ou des extensions simples au langage permet d'accroître encore plus facilement la diffusion et l'utilisation de ces outils.

3.1.3.2 Description

L'utilisation des concepts proche de ceux de l'ingénierie dirigée par un modèle m'a permis, à partir d'une spécification unique de serveurs Internet, de générer automatiquement le

serveur Internet fonctionnel mais aussi son modèle formel, vérifié par un *model checker*, en fonction du modèle de concurrence désiré [71].

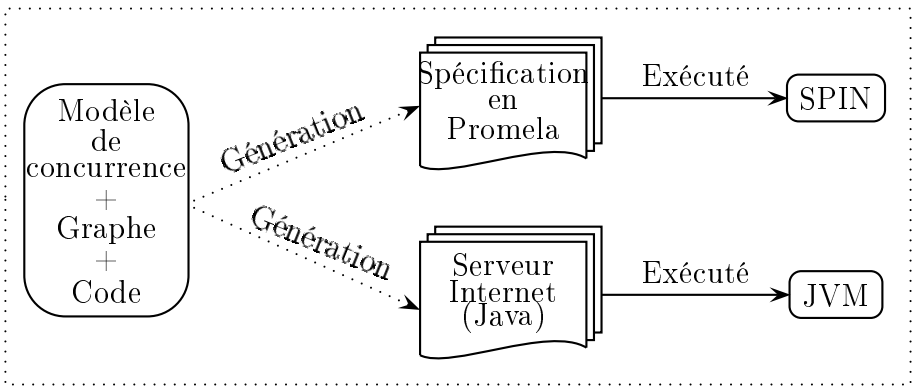


FIG. 3.3 – Processus de développement

Pour se faire, j'ai proposé un modèle de spécification de serveurs Internet qui va être constitué de trois parties spécifiées par le développeur [68, 70] :

1. un *graphe orienté* qui représente l'ensemble des opérations de synchronisation ;
2. les *nœuds du graphe* (appellés *stages*) qui fournissent le code métier ;
3. le *modèle de concurrence* qui permet de générer le code fonctionnel.

Un *stage* va représenter l'ensemble des instructions qui fournissent un traitement fondamental du serveur Internet (lecture sur une socket réseau, acceptation d'un nouveau client, calcul, etc.). Un stage va :

1. Réaliser au plus une opération d'E/S.
2. Ne contenir aucune opération de synchronisation ou de concurrence.

La phase de génération impose cette hypothèse très forte sur l'absence d'opérations de synchronisation ou de concurrence au sein des stages !

En effet, les opérations de synchronisation sont représentées par le graphe et la concurrence par le modèle de concurrence choisi par le développeur. Ce concept de *graphe bloquant* a déjà été utilisé pour l'ordonnancement, à l'exécution, des processus légers [12]. Il existe d'autres travaux qui utilisent un « graphe de communication » pour développer des applications Java concurrentes [13] ou un « automate d'E/S » pour le développement d'applications distribuées [38]. Le graphe de spécification de Saburo est une combinaison de ces approches.

Le code fonctionnel, généré selon un modèle de concurrence choisi par le développeur va réaliser le lien entre les différents stages par des appels de fonctions, des files locales ou des connexions réseaux. Il va aussi permettre de spécifier la manière d'utiliser les processus dans

l'application et le type des E/S.

L'utilisation des principes de la programmation par composants m'a permis de fournir, lors de la spécification des différents stages, un mécanisme pour les réutiliser. Ils sont fournis par des bibliothèques. La communication entre les différents stages va être spécifiée par des interfaces Java qu'il peut être nécessaire d'adapter. Pour réaliser cette adaptation, j'utilise le principe d'*héritage* fourni par les langages de programmation objet.

L'ensemble de ces mécanismes permet de construire un serveur Internet complet par simple assemblage de stages ce qui permet de réduire les coûts et les temps de développement.

3.1.3.3 Les applications comme réseaux de stages

Dans mon modèle de développement une application est construite comme un *réseau de stages*. Les stages vont être connectés *via* des files locales, des appels de fonctions ou des connexions réseau selon le modèle de concurrence utilisé.

Ce réseau de stages peut être construit :
- *statiquement* : tous les stages et leurs connexions sont connus au moment de la compilation ou du chargement de l'application ;
- *dynamiquement* : les stages peuvent être ajoutés, modifiés et supprimés lors de l'exécution de l'application.

Les deux approches présentent des avantages et des inconvénients. Ainsi la construction statique d'un réseau permet au développeur de vérifier, à l'aide d'outils dédiés, la correction de la structure du graphe, en répondant à la question :

Est-ce que les types des événements générés par un stage sont traités par un ou des stages qui sont en aval de celui-ci ?

De plus, la construction statique permet des optimisations à la compilation :
- réduction du chemin entre deux stages ;
- association de deux stages en un ;
- etc.

La construction dynamique de réseau permet une plus grande flexibilité dans la conception d'une application car il est possible d'ajouter de nouveaux stages dans un système selon les besoins. Par exemple, si une fonctionnalité est rarement appelée, le stage correspondant peut être créer à la demande.

Dans les applications performantes et à longue durée de vie, il est extrêmement important d'avoir une gestion très fine de la mémoire. Lorsqu'un objet n'est plus utilisé, celui-ci doit être libéré. Dans mon cas, l'objet sera rendu à un vivier d'objets qui permet d'amortir le coût de la création. Il est extrêmement intéressant de pouvoir calculer automatiquement le cycle de vie d'un objet pour décharger le programmeur de cette tâche. Ce mécanisme est parfaitement

assimilable au ramasse miettes du langage Java. Ici, le cycle de vie va correspondre à un sous-graphe. Ce sous-graphe va commencer du stage où l'objet est emprunté à son vivier jusqu'au stage ou il devra être relâché pour optimiser l'utilisation de la mémoire. Le graphe de stage permet de calculer le cycle de vie d'un objet. En effet, les stages vont se passer des informations entre eux. Si un objet n'est plus utilisé par la suite, il n'est tout simplement plus transmis aux stages en aval. Le calcul du cycle de vie d'un objet est réalisé en inversant le graphe. On parcours le graphe inversé en maintenant une table des objets libérés. Si un objet du stage courant n'a pas encore été libéré, on le libére et on met à jour la table des objets libérés.

Contraintes sur le graphe

Similairement aux règles de conception de circuit électrique [63], je vais définir un ensemble de contraintes sur mon graphe de synchronisation et de communication :

- *orientation* : le graphe conserve une seule orientation, d'une source vers une destination. Les boucles et les branches retours sont possibles d'un consommateur vers un producteur. On obtient ainsi *un graphe direct de flot de données.*
- *communication* : les méthodes et le format de messages de communication sont *générés automatiquement et sont spécifiques* aux composants.
- *connexion* : les stages maintiennent une connexion fixe, c'est-à-dire que les producteurs ne peuvent envoyer des messages que vers leurs consommateurs connus. Les consommateurs peuvent être ajoutés dynamiquement. La connexion est typiquement gérée en conservant les références directes entre les producteurs et les consommateurs. Cette connexion peut aussi être fournie *via* un canal partagé (une file par exemple).
- *protocole de transfert* : tous les messages portent une information mais comme un stage transmet une information mutable, il ne doit plus manipuler cette information. Dans mon modèle, le protocole de transfert est basé sur l'opération *push*.

En tenant compte de ces contraintes, on obtient différents types de stages.

Différents types de stage

Un stage *initial* n'a aucun prédécesseur et envoie des événéments vers un seul successeur (voir Fig. 3.4).

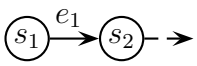


FIG. 3.4 – Le stage s_1 est un stage initial.

Un stage *final* n'a pas de successeur et reçoit des événements d'un seul prédécesseur (voir Fig. 3.5).

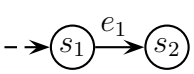


FIG. 3.5 – Le stage s_2 est un stage final.

Un stage *défaut* reçoit des événements d'un seul prédécesseur et émet des événements vers un unique successeur (voir Fig. 3.6).

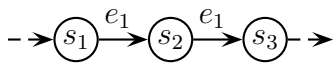


FIG. 3.6 – Le stage s_2 est un stage défaut.

Un stage *collecteur* reçoit des événements de plusieurs prédécesseurs et les retransmet vers un unique successeur (voir Fig. 3.7).

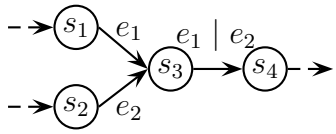


FIG. 3.7 – Le stage s_3 est un stage collecteur.

Un stage *combineur* va bloquer tant qu'il ne peut pas associer ensemble un événement de chacun de ses précédesseurs (voir Fig. 3.8).

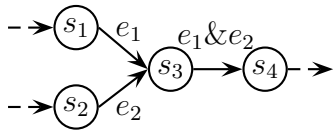


FIG. 3.8 – Le stage s_3 est un stage combineur.

Un stage *routeur* va retransmettre tout les événements obéissant à un prédicat donné vers un successeur et les autres vers un autre successeur (voir Fig. 3.9).

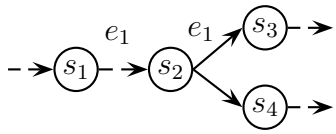


FIG. 3.9 – Le stage s_2 est un stage routeur.

Enfin, un stage *multicasteur* retransmet le même événement vers l'ensemble de ses successeurs (voir Fig. 3.10).

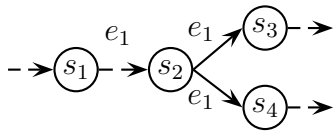


FIG. 3.10 – Le stage s_2 est un stage multicasteur.

3.2 Saburo, un outil de développement de serveurs Internet

Pour illustrer l'utilisation de mon modèle de développement, j'ai implanté Saburo un prototype en Java. Saburo est une « fabrique de serveurs Internet » qui fournit des interfaces, des classes et des annotations permettant d'implanter des stages, de les connecter dans un réseau, de gérer la configuration des serveurs ainsi que quelques stages de base réutilisables. De plus, Saburo fournit une bibliothèque de gestion des E/S non bloquantes qui est basée sur les E/S sorties fournies par le langage Java. Ces classes d'encapsulation permettent d'en faciliter l'utilisation en restreignant l'API disponible à ce contexte particulier. Dans cette section, je vais détailler la conception et l'implantation de Saburo.

La figure 3.11 illustre le principe de fonctionnement de Saburo et les interactions entre le code fourni par le développeur, les bibliothèques de Saburo, les générateurs de code source et le code généré.

3.2.1 Le langage Java

Saburo est entièrement implanté à l'aide du langage de programmation Java [40]. La décision d'utiliser ce langage au lieu d'utiliser un « langage système » (langage C [56] ou langage C++ [100]) est motivé par une très grande portabilité de Java, par la gestion automatique de la mémoire (ramasse-miettes [54]) et par la sûreté du typage. En effet, le résultat de la compilation d'un programme Java est un code intermédiaire appelé *bytecode*. Ce dernier est indépendant de la plate-forme matérielle et logicielle (que l'on soit sur un Pentium, un PowerPC, un Sparc ou sur un Alpha, sous Windows, MacOS, Solaris ou Linux, etc.). Cette indépendance garantit la portabilité des applications écrites en Java.

En général, le *bytecode* n'est pas directement exécuté par un processeur physique. Aussi, une couche logicielle est introduite entre ce *bytecode* et la machine hôte sur laquelle l'exécution doit se dérouler. Cette couche logicielle est appelée *machine virtuelle Java*. Sa principale fonction est d'exécuter les séquences d'instructions de *bytecode*. La sémantique de chaque instruction de bytecode est décrite dans la spécification de la machine virtuelle Java fournie par Sun [66]. Un programme Java n'est compilé qu'une fois pour toutes mais son *bytecode* peut être exécuté sur n'importe quel système (*« Write once, run anywhere »*), pourvu que ce dernier possède sa propre implantation de machine virtuelle Java (voir Fig. 3.12).

L'ensemble des caractéristiques du langage Java permet une plus grande robustesse des applications développées. De plus, le langage Java permet d'éviter une large classe de problèmes courant en programmation :

- violation des limites d'un tableau ;
- erreurs lors de la libération d'une zone mémoire ;
- utilisation d'un mauvais type ;
- etc.

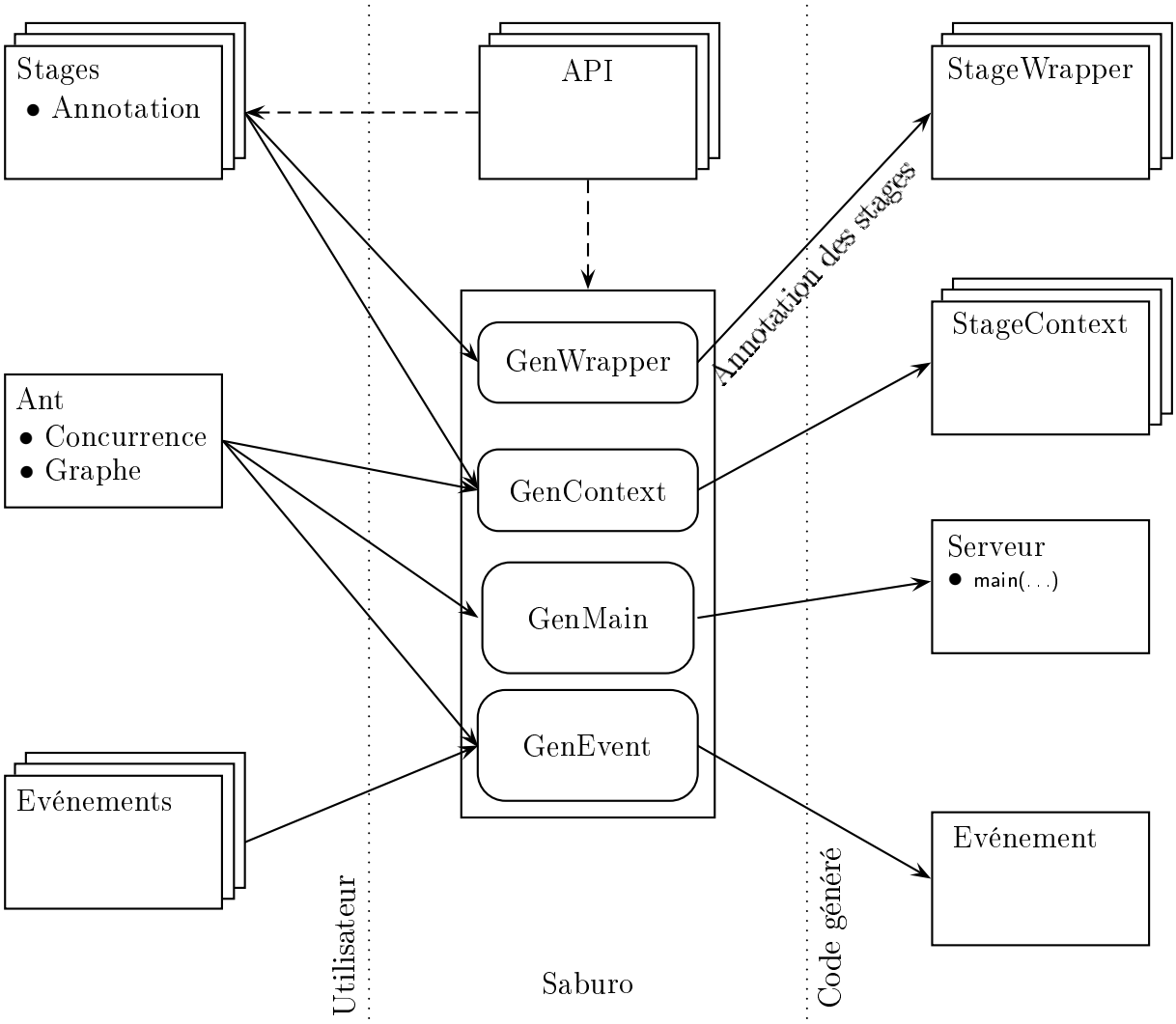


FIG. 3.11 – Vue générale de Saburo.

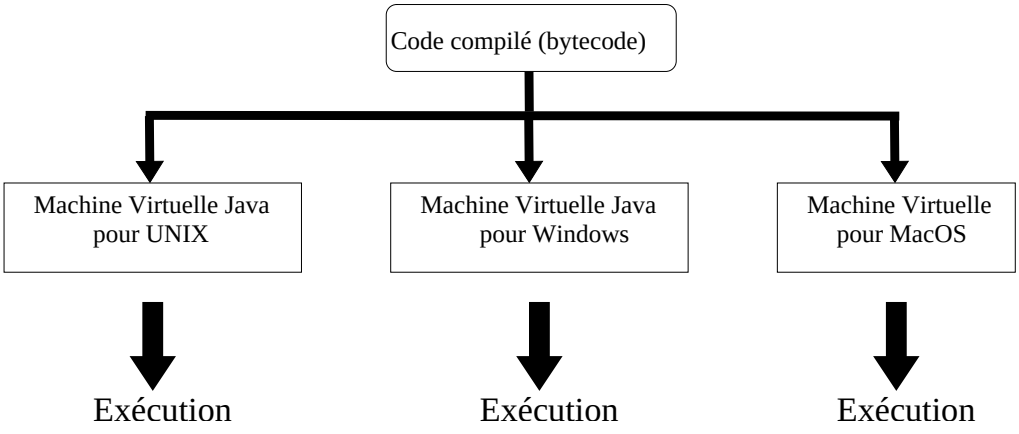


FIG. 3.12 – Exécution *via* une machine virtuelle

Je considère que la sûreté et les bénéfices en conception d'applications induit par le langage Java (i) programmation objet, (ii) introspection, (iii) génération de bytecode à la volée, sont des critères plus important que la perte de performances a fortiori de plus en plus minimes.

Cependant, l'un des principaux inconvénients du langage Java était l'absence d'E/S asynchrones. Ces E/S permettent notamment d'utiliser un unique processus et non plus un processus par connexion ou par requête. Depuis la version 1.4 du langage, il existe un support pour de telles E/S [104] et la bibliothèque NBIO [109] permet d'utiliser les E/S non bloquantes pour des versions antérieures à la version 1.4.

3.2.2 Stages et graphe de stages dans Saburo

Dans cette section, je vais tout d'abord présenter les stages standards fournis par Saburo. Fournir des stages réutilisables permet aux développeurs d'accélérer le développement de leurs serveurs Internet. Dans un second temps, je vais introduire les classes nécessaires à la spécification du graphe de stages d'un serveur. Et enfin, je vais présenter la génération du code dans mon outil.

3.2.2.1 Stages fournis « sur étagère » par Saburo

Chaque stage va implanter une seule méthode handle(...) qui représente les instructions à exécuter. Ses paramètres vont être un contexte, utilisé pour accéder au(x) successeur(s), potentiellement un événement d'entrée et/ou un événement de sortie. L'implantation des différents événements est générée automatiquement à l'aide du générateur GenEvent. La présence d'événements d'entrée ou de sortie dépend de la position du stage dans le graphe. Le contexte va correspondre au canal de connexion entre le stage et son ou ses successeur(s) et il peut être généré comme un appel de fonction, une file locale ou une connexion réseau par le générateur GenContext. Un stage possédant un ou des successeur(s) va donc utiliser son contexte pour leur envoyer le ou les événement(s) de sortie grâce à la méthode dispatchToSuccessor(...).

Saburo fournit trois stages standards accompagnés de leurs événements de communication. Le premier est le stage Accept qui permet d'accepter un client sur une connexion serveur.

```
@Initial
@Stage
public class AcceptStage< S extends AcceptStageResponse > {
  private final SaburoServerSocket server;
  public AcceptStage(Configuration configuration) throws IOException {
    server = configuration.getSaburoServerSocket(("acceptStage.SaburoServerSocket"));
  }

  public final void handle(StageContext< S > ctx, S res) {
    SaburoSocket socket;
    if((socket = server.accept()) != null) {
      res.setSaburoSocket(socket);
```

```
      ctx.dispatchToSuccessor(res) ;      }
  }
}
```

Afin de créer l'objet SaburoServerSocket, il est nécessaire d'avoir le nom du serveur et son port. Ces informations sont spécifiées dans un fichier de configurations par le développeur. L'objet Configuration va lire un fichier de configuration et va créer les objets adéquats tel que l'objet SaburoServerSocket. Le constructeur de l'objet AcceptStage prend en paramètre cet objet de configuration pour récupérer les informations qui lui sont nécessaires, i.e. l'objet SaburoServerSocket. L'annotation @Initial est nécessaire car elle permet d'indiquer à Saburo que le stage accept est un stage initial. Comme c'est un stage initial, il n'a pas d'événement en entrée ce qui permet de faire des vérifications statiques sur la structure du graphe lors de la génération par le générateur GenMain et ainsi améliorer la sûreté de l'application générée.

Le second stage fourni par Saburo est le stage Read qui permet de lire des données sur une connexion cliente.

```
@Stage
public class ReadStage< Q extends ReadStageRequest, S extends ReadStageResponse > {
  public final boolean handle(StageContext< S > ctx, Q req, S res) {
    RequestStream stream = req.getRequestStream() ;
    ByteBuffer buffer = ctx.takeByteBuffer(res) ;
    int n = 0 ;
    try {
      if((n = stream.read(buffer)) > 0) {
        buffer.flip() ;
        res.setByteBufferRequest(buffer) ;
        ctx.dispatchToSuccessor(res) ;
      } else if(n == -1) {
        ctx.backByteBuffer(buffer) ;
        return false ;
      }
    } catch(IOException e) {
      stream.shutdown() ;
      ctx.backByteBuffer(buffer) ;
      return false ;
    }
    return true ;
  }
}
```

Dans ce stage la socket est fermée logiquement en lecture par la méthode shutdown(), i.e. il n'y aura plus de lecture possible sur la connexion cliente. L'annotation @Stage indique que le stage *read* reçoit un événement en entrée et en envoie un en sortie vers son successeur. Ces informations sont utilisées par les générateurs GenMain et GenContext.

Ce stage est complètement indépendant du type d'E/S. Dans le cas d'E/S bloquantes il est nécessaire d'appeler cette méthode handle() de la façon suivante :

```
while(read.handle(context, in, out)) ;
```

Dans le cas d'E/S non bloquantes le mécanisme des sélecteurs aura la charge d'appeler cette méthode.

Remarque : Le code d'appel à la méthode handle(...) est généré automatiquement par les générateurs GenWrapper, GenMain et GenContext selon le modèle de concurrence.

Enfin, le dernier stage fourni par Saburo est le stage Write qui permet d'écrire des données sur une connexion cliente.

```
@Stage
@Final
public class WriteStage< Q extends WriteStageRequest > {
  public int handle(StageContext< ? > ctx, Q req) {
    ResponseStream stream = req.getResponseStream() ;
    ConcurrentLinkedQueue< ByteBuffer > buffers = req.getByteBufferResponse() ;
    if(buffers.size() > 0) {
      ByteBuffer buffer = buffers.peek() ;
      if(buffer.hasRemaining() == false) {
        buffers.poll() ;
        ctx.backByteBuffer(buffer) ;
      } else {
        try {
          stream.write(buffer) ;
        } catch(IOException e) {
          while(buffers.size() > 0) {
            ctx.backByteBuffer(buffers.poll()) ;
          }
          return -1 ;
        }
        if( !buffer.hasRemaining()) {
          buffers.poll() ;
```

```
            ctx.backByteBuffer(buffer) ;
          }
        }
        return 1 ;
      }
      return 0 ;
    }
  }
```

Ce stage est un peu plus compliqué car on doit prendre en compte le cas non bloquant. En effet, les données qui sont stockées dans des ByteBuffer peuvent ne pas être écrites totalement. Il faut tester ce cas particulier. C'est pourquoi, ce stage maintient une liste (pour conserver l'ordre d'écriture) des tampons à écrire, emprunte sans le retirer un tampon et essaye de l'écrire. Si le tampon est entièrement écrit alors le tampon est retiré de la liste puis libéré, sinon le tampon est laissé dans la liste pour finir de l'écrire à un autre moment.

3.2.2.2 Graphe des stages

Pour la gestion du graphe, j'utilise le principe de génération de programme dirigée par des annotations [91]. Les stages sont ainsi « marqués » en utilisant les 7 annotations possibles (Initial, Final, Stage, Router, Multicaster, Combiner, Collector). Cependant en Java il n'est pas possible d'avoir de relation d'héritage entre annotations. Ainsi, un stage routeur aura deux annotations. La première indiquant au gestionnaire de stage que c'est un stage (annotation Stage) et la seconde qui indique que le stage est un routeur (annotation Router).

La connexion des stages est ensuite spécifiée à la main sous forme de code Java ou bien à l'aide d'une tâche Ant et du générateur GenMain pour automatiser plus facilement la construction du serveur. Les stages sont connectés *via* une implantation de l'interface StageManager. Cela permet de rajouter des vérifications à la compilation, par exemple :

1. Existe t-il au moins un stage initial et un stage final ?
2. Les stages connectés ensemble sont-ils bien compatibles ?
3. etc.

L'interface StageManager fournit les méthodes suivantes :

```
public interface StageManager {
  public Iterator< String > initialStages() ;
  public Iterator< String > next(String id) ;

  public Class< ? > getStage(String id) ;

  public boolean isInitialStage(String id) ;
  public boolean isFinalStage(String id) ;
```

```
  public void connect(Class< ? > source, Class< ? > sink) ;
  public void connect(Class< ? > source, String sourceName, Class< ? > sink, String sinkName) ;
}
```

La méthode initialStages() permet de récupérer l'ensemble des stages initiaux d'un graphe. Cela permet d'initier le parcours du graphe lors de la génération du code dépendant du modèle de concurrence (lors de la génération des contextes avec GenContext et des *wrappers* avec GenWrapper). La méthode next() permet de récupérer les successeurs du stage que l'on donne en paramètre. A chaque stage est associée une clé unique qui permet de récupérer la classe d'implantation du stage pour appliquer des opérations d'introspection (récupérer le constructeur, des méthodes, etc.). Les deux méthodes connect() permettent de connecter deux stages ensemble et de vérifier leur compatibilité.

Evénements de communication

Lors du développement, le développeur va spécifier les événements de communication entre les différents stages. Ces événements sont spécifiés sous forme d'interfaces Java. J'ai choisi d'utiliser la convention de nommage suivante pour ces interfaces :

<type><StageName>Event

Le <type> vaut soit Request soit Response et <StageName> correspond au nom du stage envoyant l'événement. Par exemple, l'événement ResponseAcceptEvent est l'événement sortie du stage Accept.

Ces interfaces vont seulement déclarer des méthodes (send() or receive()) qui sont utilisées par les stages. Le stage en amont va envoyer des données et le stage en aval va recevoir des données. L'implantation de ces méthodes est générée automatiquement durant la phase de génération par le générateur GenEvent. Ces méthodes suivent la convention de nommage suivante :

<method><StageName><DataType>

Avec <method> qui vaut soit send soit receive, <StageName> correspond au nom du stage envoyant l'événement et DataType correspond au type des données envoyées ou reçues. Par exemple, la méthode sendAcceptServerSocket() indique que la méthode permet d'envoyer du stage Accept une ServerSocket.

Ces interfaces sont ensuite fournies en entrée au générateur d'événements (GenEvent). Ce générateur va produire une unique implantation de toutes ces interfaces. Cela permet au processus de génération de choisir la meilleure implémentation des événements selon le modèle de concurrence choisi (rajouter des informations ou en supprimer).

Remarque : Ce mécanisme permet de libérer le développeur de la création des événements et de l'implantation du vivier d'objets qui lui aussi est généré par GenEvent. Une optimisation des générateurs, qui n'est pas encore implantée, est la détection des cycles de vie des différents objets encapsulés par les événements. Cela permet de libérer « au bon moment » les objets qui ne seront plus utilisés par la suite. Une idée pour détecter ce cycle de vie est d'inverser le graphe et de le parcourir pour détecter à partir de quel stage un objet n'est plus utilisé. C'est-à-dire que l'on détecte à quel moment il n'y a plus de méthode qui récupére cet objet dans les interfaces d'événements. Détecter leur « création » correspond à détecter la méthode fixant cet objet dans une interface d'événement.

3.2.2.3 Utilisation des annotations

Les annotations sont utilisées pour marquer le type du stage selon les patrons définis dans la section 3.1.3.3. Cela me permet de vérifier lors du processus de génération :

1. Qu'il existe au moins un stage initial et un stage final dans le graphe.
2. Que les objets que j'essaye de connecter ont bien été conçus pour Saburo.
3. que je n'essaye pas de connecter un stage final (pas de successeur) à un stage initial (pas de prédecesseur).
4. *etc.*

Par la suite les annotations vont être utilisées dans le processus de vérification pour extraire des informations pertinentes du serveur Internet (plus particulièrement sur sa structure).

Afin de modèliser des applications concurrentes beaucoup plus complexes, les annotations peuvent être composées. La composition se fait en énumérant les annotations que le stage va respecter. Ainsi si l'on souhaite qu'un stage puisse collecté des événements (annotation Collecter) et les envoyés vers l'ensemble de ses successeurs (annotation Multicaster), on le déclarera de la façon suivante :

```
@Stage
@Collector
@Multicaster
public class ComposeStage<S extends ComposeStageResponse, Q extends ComposeStageRequest > {
}
```

Le code adéquat sera ensuite généré par les générateurs :

- GenContext pour l'annotation Multicaster car le contexte représente tout les traitements en aval d'un stage.
- GenWrapper pour l'annotation Collecter car les « wrappers » représentent tout les traitements en amont d'un stage.

Remarque : L'annotation Inital (respectivement Final) ne pourra pas être composée avec les annotations Collector et Combiner (respectivement Router et Multicaster).

3.2.2.4 Générateurs de code

Saburo fournit un ensemble de générateurs (GenWrapper, GenContext, GenMain et GenEvent) utilisés pour générer les serveurs Internet en fonction du graphe de spécification et du modèle de concurrence choisi par le développeur.

GenWrapper :

Description : Ce générateur permet de générer le *wrapper* de chaque stage, *i.e* l'ensemble des traitements réalisés en amont du stage.

Entrées : Il prend en entrée un stage et plus particulièrement son annotation afin de générer du code d'encapsulation permettant :

- La combinaison d'événements en entrée.
- La collection d'événements en entrée.

Sorties : Ce générateur va produire le code exécuté en amont du stage (appelé *wrapper*).

GenContext :

Description : Ce générateur permet de générer le *contexte* de chaque stage. Le contexte est utilisé pour échanger des informations avec les successeurs du stage et regroupe l'ensemble des traitements réalisés en aval du stage.

Entrées : Il prend en entrée un stage, son annotation, le graphe de spécification du serveur Internet et le modèle de concurrence. Selon le modèle de concurrence et le graphe, il va générer le code permettant de communiquer avec les successeurs du stage. Dans le cas d'une approche par compétition, la communication se fait par des appels de fonctions. Pour les approches par coopération, des files locales sont utilisées. Selon l'annotation du stage, il va aussi générer du code d'encapsulation permettant :

- Le routage d'événements en sortie.
- Le *multicast* d'événements en sortie.

Sorties : Ce générateur va générer le code exécuté en aval d'un stage (appelé *contexte*).

Remarque : A l'aide du *wrapper* et du *contexte*, on peut calculer très facilement la latence d'un stage en utilisant un compteur initialisé dans le *wrapper*. Cela permet de détecter les stages à optimiser.

GenMain :

Description : Ce générateur va produire la boucle principale du serveur, les instanciations d'objets nécessaires au serveur, *etc.*

Entrées : Il prend en entrée le graphe de communication et le modèle de concurrence. A partir du graphe, il va produire l'instanciation de tous les objets nécessaires au serveur Internet (stages, contextes, *wrappers*) en le parcourant dans l'ordre inverse. Puis en fonction du graphe et du modèle de concurrence, il va produire la boucle principale.

Sortie : Ce générateur permet de produire la fonction main(...) du serveur Internet. Cette fonction va réaliser l'analyse d'un fichier de configuration, instancier l'ensemble des stages, contextes et *wrappers* (dans l'ordre inverse du graphe). Puis, elle va exécuter le code associé au modèle de concurrence choisi par le développeur.

GenEvent :

Description : Ce générateur permet de produire le code des événements à partir des interfaces déclaratives spécifiées par le développeur et du modèle de concurrence.

Entrées : Il prend en entrée les différentes interfaces déclaratives ainsi que le modèle de concurrence choisi par le développeur. Le modèle de concurrence est utile ici car les différents décorateurs des stages (*wrapper* et/ou *contexte*) vont s'échanger des informations induites par le modèle de concurrence. Mon but étant de m'abstraire du modèle de concurrence lors de la spécification d'un serveur Internet, ses informations d'ordre implantatoire vont être générées automatiquement.

Sortie : Ce générateur va produire l'implantation des différentes interfaces déclaratives utilisées pour la communication inter-stages.

La figure 3.13 modèlise les contextes et les *wrappers* d'une partie du serveur HTTP généré par Saburo. Ils sont produits respectivement par les générateurs GenContext et GenWrapper.

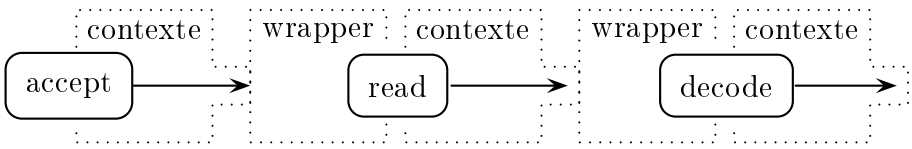


FIG. 3.13 – Modèlisation d'une partie du code généré d'un serveur HTTP

Le code généré peut être en Java ou en bytecode. Le bytecode est produit à l'aide d'ASM [20]. Cette seconde approche permet de modifier « à la volée » un serveur Internet et de rajouter des stages dynamiquement pour fournir un nouveau service ou enrichir un service déjà existant. Dans mon API, les générateurs vont implanter l'interface Generator :

```
public interface Generator< I extends AbstractBean, O > {
  public void generate(I input, O output) throws GeneratorException;
}
```

Cette interface génére du code à partir d'informations réunies dans un *bean* [105]. Le *bean* sert à stocker toutes les informations nécessaires à la génération d'un objet (paquetage utilisé, classes à importer, événements utilisés par un stage, successeurs, etc.). Le *bean* est rempli à partir du graphe et il est utilisé comme point d'entrée du processus de génération. Le stage en lui-même n'est pas modifié par le processus de génération. On va lui adjoindre une classe d'encapsulation permettant de communiquer avec ses successeurs : le *contexte* du stage. De plus, le processus de génération va produire la boucle principale du serveur.

3.2.3 Interface de programmation de Saburo

Je vais maintenant détailler les classes qui sont utilisées pour faciliter le développement d'un serveur Internet. Ces classes sont des implantations de viviers d'objets, de processus ou de caches de fichiers.

Viviers d'objets

Dans un serveur Internet, la gestion mémoire est une préoccupation extrêmement importante car, la création d'un objet est une opération très coûteuse. Pour éviter cette perte de temps, il est recommandé de créer les objets au moment de l'initialisation du serveur puis de les réutiliser en utilisant un mécanisme de *viviers*. Dans Saburo, les objets qui peuvent être gérés par un vivier implantent l'interface `PoolEntry` :

```
public interface PoolEntry {
  public void clear();
}
```

La méthode `clear()` est utilisée pour réinitialiser un objet lorsqu'il est relâché dans le vivier. Les différents viviers de processus implantent l'interface `Pool` :

```
public interface PoolManager< E extends PoolEntry > {
  public void back(E entry);
  public abstract E take();
}
```

La méthode back() est utilisée pour relâcher un objet dans le vivier et la méthode take pour emprunter un objet au vivier. La création des objets est assurée par une fabrique qui implantent l'interface Factory :

```
public interface Factory< E extends PoolEntry > {
  public E newInstance() ;
}
```

La méthode newInstance() permet de créer un objet qui sera ensuite conservé dans un vivier. Cette méthode est systématiquement appelée lors de l'initialisation du vivier. De plus et suivant la stratégie du vivier, cette méthode pourra être appelée afin d'augmenter le nombre d'objets présents dans le vivier.

Cache de fichiers

Dans le cas d'un serveur HTTP, le client va demander au serveur des fichiers. Le serveur va devoir lire le fichier, puis l'écrire dans une connexion réseau. Ces opérations consistent à charger le fichier en mémoire (*sérialiser* le fichier) puis de le réécrire physiquement (*désérialiser* le fichier). Pour « gagner » une étape on peut conserver un fichier en mémoire dans un *cache de fichiers*. Dans Saburo, les caches de fichiers implantent l'interface FileCache :

```
public interface FileCache {
  public void put(String path) throws IOException;
  public ByteBuffer get(String path);
  public long getSize(String path);
}
```

Un cache est implanté comme une table de hachage. La méthode :

- put() sert à conserver dans le cache le fichier qui a le chemin spécifié en paramètre ;
- get() permet de récupérer une vue sérialisée en mémoire d'un fichier ;
- getSize() permet de récupérer la taille du fichier. Cette méthode permet de remplir le champ *content-Length* de la réponse du serveur HTTP destinée au client.

La sérialisation d'un fichier se réalise à l'aide de classe MappedByteBuffer qui permet de « monter » un fichier en mémoire.

3.3 Développement d'un serveur HTTP avec Saburo

Je vais maintenant présenter le développement d'un serveur Internet produit à l'aide de Saburo. Dans un service Internet, l'une des préoccupations principales est de *répondre à un*

maximum de clients en un minimum de temps mais aussi d'être *robuste face aux montées en charge importantes*. Une seconde préoccupation est d'optimiser l'utilisation des ressources systèmes en fonction des besoins et de leur disponibilité. En effet, un serveur Internet peut être utilisé pour configurer un pda ou répondre aux requêtes de milliers de clients sur une machine multi-processeurs. Les ressources disponibles et utilisées vont alors être extrêmement variables.

Les serveurs HTTP sont typiquement des serveurs Internet qui doivent être robustes face aux montées en charge très brusques et la plupart des travaux de recherche se consacrent principalement à l'optimisation des performances. Cependant, très peu s'intéressent à la simplification de leur développement ou à leur adaptation en fonction des ressources strictement nécessaires ou disponibles. Je vais ainsi décrire le développement d'un serveur HTTP simple pour illustrer le développement d'une application significative, complète et réelle.

L'un des intérêts d'étudier les serveurs HTTP est l'existence d'une très grande variété d'outils permettant de mesurer leurs performances [5, 45] ainsi qu'un grand nombre de résultats publiés comparant les performances de différentes architectures et/ou serveurs HTTP.

3.3.1 HTTP, un protocole sans état

Un protocole sans état est un protocole qui ne maintient aucune relation entre les différentes requêtes d'un même client, i.e. les requêtes sont considérées comme indépendantes les unes des autres. Le protocole HTTP (HyperText Transfer Protocol) est l'exemple le plus courant de protocole sans état. En effet, entre différentes requêtes, un serveur HTTP ne va jamais maintenir et réutiliser d'informations (nom d'utilisateur, mot de passe, etc.) sur ses clients.

Lors de la réception d'une requête, un serveur HTTP va se comporter de la façon suivante :

1. il crée et active un processus de traitement ou en réactive un (s'il maintient un vivier de processus de traitement) pour servir la requête d'un client ;
2. ce processus traite la requête et génére une réponse qui est envoyée au client ;
3. le processus de traitement est tué ou mis en sommeil (s'il y a utilisation d'un vivier de processus).

La nature sans état du protocole HTTP implique qu'un processus de traitement ne doit jamais être alloué de façon permanente à un client particulier. En effet, après avoir traité une requête, un processus de traitement va potentiellement servir plusieurs requêtes de clients distincts avant de recevoir une autre requête du client originel, s'il y en a une. Ainsi, lorsque le serveur reçoit la requête d'un client, il n'est pas nécessaire de mettre en place une technique particulière et coûteuse garantissant l'attribution du même processus de traitement ayant déjà traité les requêtes précédentes de ce client.

Remarque : Du fait de la nature sans état du protocole HTTP, un processus de traitement mis en sommeil doit oublier toutes les informations sur la requête qu'il vient de traiter

et la réponse qu'il a générée. Ainsi, si la requête suivante d'un même client est donnée en traitement à ce processus, il n'aura aucun « souvenir » des traitements effectués pour la requête précédente et respectera la nature sans état du protocole HTTP.

L'absence d'état d'un protocole permet de ne pas limiter le nombre de processus de traitement à attribuer aux connexions entrantes. En effet, dans un protocole sans état n'importe quel processus de traitement peut être attribué à n'importe quelle requête. *A contrario*, dans le cas d'un protocole avec état un processus va conserver les informations sur une connexion tant que celle-ci n'est pas fermée. Il ne peut pas être attribuer à une autre connexion ce qui limite le nombre de processus de traitement assignable. Comme aucune information issue des activités précédentes des clients n'est conservée, chaque requête est traitée comme une nouvelle requête et n'est pas considérée comme une requête parmi un flot de requêtes.

3.3.2 Développement d'un serveur HTTP

Je vais maintenant présenter le développement d'un serveur HTTP simple. Cet exemple va permettre d'illustrer entièrement le processus de développement à l'aide de mon outil Saburo. Plus précisément, le processus de développement va être composé de quatre étapes :

1. la spécification du graphe de communication/synchronisation ;
2. la spécification des événements de communication ;
3. le développement des stages ;
4. la phase de génération.

3.3.2.1 Spécification du serveur HTTP

La première étape du processus de développement consiste à découper le serveur Internet en plusieurs stages. Ce découpage va dépendre des opérations de communication et de synchronisation réalisées dans le serveur, mais aussi d'une décomposition logique de l'application.

Le serveur HTTP pris en exemple va être composé de six stages : (i) le stage *accept* qui accepte les nouveaux clients et établit les connexions, (ii) le stage *read* qui lit les données reçues sur une connexion, (iii) le stage *decode* qui interprète, i.e. *désérialise*, la requête reçue d'un client, (iv) le stage *service* qui charge le fichier demandé depuis le disque dur du serveur, si celui-ci existe, (v) le stage *encode* qui code, i.e. *sérialise*, la réponse pour le client, enfin (vi) le stage *write* qui écrit la réponse pour le client.

Le graphe direct représenté par la figure 3.3.2.1 illustre ce découpage et les interconnections de ces six stages.
Ce graphe est spécifié par le développeur *via* le code Java suivant :

```
StageManagerImpl manager = new StageManagerImpl() ;
manager.connect(AcceptStage.class, ReadStage.class) ;
```

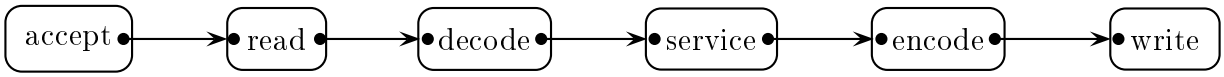


FIG. 3.14 – Le service HTTP et ses différents stages

```
manager.connect(ReadStage.class, DecodeStage.class);
manager.connect(DecodeStage.class, ServiceStage.class);
manager.connect(ServiceStage.class, EncodeStage.class);
manager.connect(EncodeStage.class, WriteStage.class);
```

Ce code peut être généré automatiquement par GenMain à partir d'une tâche Ant qui spécifie entre autre les connexions entre les stages. Un exemple de spécification du graphe *via* une tâche Ant est donnée page 79.

Remarque : Les stages *accept*, *read* et *write* sont des stages génériques qui sont réutilisables par d'autres spécifications de serveurs. Les stages *decode*, *service* et *encode* sont des stages spécifiques au serveur HTTP.

3.3.2.2 La spécification des événements de communication

Afin d'échanger des données et selon la position du stage dans le graphe, des événements d'entrée et sortie sont spécifiés pour, respectivement, recevoir et envoyer des données. Ces *événements* sont spécifiés *via* des interfaces Java. Je vais maintenant illustrer la spécification des événements de mon serveur HTTP (voir Fig. 3.3.2.1).

Evénement du stage *accept*

Une seule interface pour les événements de sortie.

```
public interface OutputAcceptEvent {
  public void setSaburoSocket(SaburoSocket s);
}
```

Le stage *accept* accepte un client, établit une nouvelle connexion cliente et envoie le point d'entrée de cette connexion (la *socket*) aux stages *read* et *write*.

Remarque : La socket cliente est directement envoyée au stage *read* (ligne pleine de la figure 3.15) et indirectement au stage *write* (ligne pointillée de la figure 3.15).

Evénements du stage *read*

Un événement d'entrée et un événement de sortie.

```
public interface InputReadEvent {
  public SaburoSocket getSaburoSocket();
}
```

```
public interface OutputReadEvent {
  public void setRequestByteBuffer(ByteBuffer b);
}
```

Le stage *read* lit des données depuis la socket cliente. L'API de Saburo fournie pour la lecture et l'écriture de données est basée sur le paquetage java.nio qui fournit des E/S bloquantes et non bloquantes. Ainsi, la classe SaburoSocket va fournir des traitements de lecture d'une séquence d'octets stockée dans un ByteBuffer depuis une SocketChannel qu'elle encapsule. Le stage *read* envoie ensuite le ByteBuffer au stage *decode*.

Evénements du stage *decode*

Un événement d'entrée et un événement de sortie.

```
public interface InputDecodeEvent {
  public ByteBuffer getRequestByteBuffer();
  public SaburoSocket getSaburoSocket();
}
```

```
public interface OutputDecodeEvent {
  public void setReqMethod(HttpMethod met);
  public void setReqUri(URI uri);
  public void setReqVersion(Version vers);
  public void setReqContentLength(int length);
}
```

Le stage *decode* reçoit un ByteBuffer du stage *read*. Il va décoder la séquence d'octets afin d'obtenir la requête HTTP du client. Le but du stage *decode* est de générer la représentation logique de la séquence d'octets (la requête) depuis une représentation physique (le ByteBuffer). La figure 3.15 représente la relation entre les stages, les données échangées entre eux et le type de représentation des données (physique, comme une séquence d'octets ou logique, comme une requête ou une réponse HTTP).

Stages restants

Evénements d'entrée et de sortie du stage *service* :

```
public interface InputHttpServiceEvent {
  public HttpMethod getRequestMethod();
  public URI getRequestUri();
  public Version getRequestVersion();
  public int getRequestContentLength();
}
```

```
public interface OutputHttpServiceEvent {
  public void setResponseVersion(Version v);
  public void setResponseCode(StatusCode c);
  public void setResponseMsg(StatusMsg m);
}
```

Evénements d'entrée et de sortie du stage *encode* :

```
public interface InputEncodeEvent {
  public Version getResponseVersion();
  public StatusCode getResponseStatusCode();
  public StatusMsg getResponseStatusMsg();
}
```

```
public interface OuputEncodeEvent {
  public void setResponseByteBuffer(ByteBuffer
b);
}
```

Evénement d'entrée pour le stage *write* :

```
public interface InputWriteEvent {
  public SaburoSocket getSaburoSocket();
  public ByteBuffer getResponseByteBuffer();
}
```

La figure 3.3.2.3 représente les différents passages d'informations entre les stages. Ce passage peut être direct entre deux stages (lignes pleines) ou indirectes (lignes pointillées). Cette figure montre aussi les différents niveaux de représentation des données, i.e. données physiques (les octets qui sont lus ou écrits) et les données logiques (les objets représentant les requêtes ou les réponses).

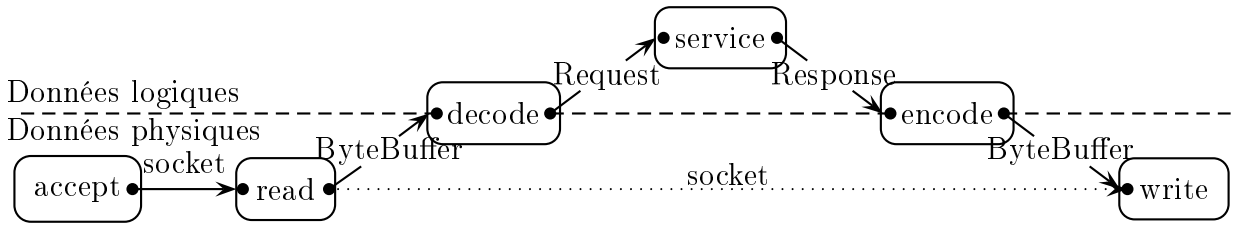


FIG. 3.15 – Passage d'informations entre les différents stages

3.3.2.3 Spécification des différents stages

Une fois que les événements de communication ont été spécifiés par le développeur, il va devoir implanter ses différents stages qui correspondent au code métier de son serveur. Je rappelle qu'un stage est une suite d'instructions avec au plus un appel d'E/S, sans aucun bloc de synchronisation.

Implantation du stage *decode*

Ce stage est beaucoup plus compliqué que ceux précédemment présentés. Sa méthode handle(...) doit être spécifiée par le développeur car elle dépend du protocole de communication utilisé (ici HTTP).

```
@Stage
public class DecodeStage< Q extends DecodeStageRequest, S extends DecodeStageResponse > {
  public final void handle(StageContext< S > ctx, Q req, S res) {
    HttpSaburoLexer lexer = req.getHttpSaburoLexerRequest() ;

    while(lexer.hasRemaining())
      lexer.step() ;

    lexer.compact() ;

    if(lexer.endOfParsing()) {
      ctx.dispatchToSuccessor(res) ;
      lexer.reset() ;
    }
  }
}
```

L'événement en entrée du stage *decode* va fournir un analyseur syntaxique pour décoder la requête cliente. L'analyseur syntaxique est fourni par le développeur et doit tenir compte du type des E/S (voir section 3.3.3). La méthode dispatchToSuccessor() est appellée lorsque l'on atteint un marqueur de fin d'analyse. Ainsi, dans le cas du protocole HTTP version 1.1, il est possible de mettre ce marqueur à la fin de la ligne de requête. Ce qui permet d'amorcer la chaîne de traitements (récupération du fichier dans le cache, écriture de celui-ci, etc.) tout en finissant d'analyser la requête.

Implantation des stages restants

Je vais maintenant décrire succinctement les différents stages restants. Les stages *service* et *encode* sont spécifiés par le développeur car ils dépendent du serveur. Le stage *write* est un stage fourni par Saburo.

```
@Stage
public class HttpServiceStage< Q extends HttpServiceStageRequest, S extends HttpServiceStageResponse > {
  private final void getMethod(String url, StageContext< S > ctx, HttpServiceStageResponse res) {

    try {
      ctx.putInCacheFile(url) ;
      res.setStatusResponse(Status.OK_STATUS) ;
      res.setIdInFileCacheResponse(url) ;
    } catch(IOException e) {
      res.setStatusResponse(Status.NOT_FOUND_STATUS) ;
    }
  }

  public final void handle(StageContext< S > ctx, Q req, S res) {
    res.setVersionResponse(Version.HTTP_1_1) ;

    switch(req.getMethodRequest()) {
    case GET :
      getMethod(req.getUrlRequest(), ctx, res) ;
    }

    ctx.dispatchToSuccessor(res) ;

  }
}
```

```
@Stage
public class EncodeStage< Q extends EncodeStageRequest, S extends EncodeStageResponse > {

  private final void writeStatusLine(ByteBuffer buffer, StageContext< S > ctx, Q req, S res) {
    int major = req.getVersionResponse().getMajorVersion() ;
    int minor = req.getVersionResponse().getMinorVersion() ;

    req.getStatusResponse().write(buffer, major, minor) ;

    switch(req.getStatusResponse()) {
    case OK_STATUS :
      Header.CONTENT_LENGTH_HEADER.write(buffer) ;
      long size = ctx.getLengthFromCacheFile(req.getIdInFileCacheResponse()) ;
      ByteBufferUtil.putLong(buffer, size) ;
      Header.CRLF_HEADER.write(buffer) ;

      Header.KEEP_ALIVE.write(buffer) ;
      Header.CRLF_HEADER.write(buffer) ;

      Header.CRLF_HEADER.write(buffer) ;
      break ;
    }
    buffer.flip() ;
    res.setByteBufferResponse(buffer) ;
  }

  public final void handle(StageContext< S > ctx, Q req, S res) {
    ByteBuffer buffer = ctx.takeByteBuffer(null) ;

    writeStatusLine(buffer, ctx, req, res) ;

    switch(req.getStatusResponse()) {
```

```
    case OK_STATUS :
      res.setByteBufferResponse(ctx.getFromCacheFile(req.getIdInFileCacheResponse())) ;
      ctx.dispatchToSuccessor(res) ;
    }
  }
}
```

3.3.2.4 Génération automatique du code technique

La phase de génération du code prend en entrée les spécifications précédentes et en fonction du modèle de concurrence choisi par le développeur va produire automatiquement l'implantation des événements (GenEvent), les contextes (GenContext) et la boucle principale du serveur (GenMain). Le modèle de concurrence est choisi par le développeur parmi les 6 modèles de concurrence pris en compte par Saburo. Lors de la génération, il est spécifié par une tâche ANT de la façon suivante :

```
<target name="seda" depends="compile">
        <model type="seda" destination="$gen-src" generatorType="java" manager="fr.umlv.saburo.core.impl.StageManagerImpl">
    <connect source="fr.umlv.saburo.core.stage.AcceptStage" sink="fr.umlv.saburo.core.stage.ReadStage" />
    <connect source="fr.umlv.saburo.core.stage.ReadStage" sink="fr.umlv.saburo.samples.http.stage.DecodeStage" />
    <connect source="fr.umlv.saburo.samples.http.stage.DecodeStage"
             sink="fr.umlv.saburo.samples.http.stage.HttpServiceStage" />
    <connect source="fr.umlv.saburo.samples.http.stage.HttpServiceStage"
             sink="fr.umlv.saburo.samples.http.stage.EncodeStage" />
    <connect source="fr.umlv.saburo.samples.http.stage.EncodeStage" sink="fr.umlv.saburo.core.stage.WriteStage" />
    <package model="$package.seda" />
  </model>
</target>
```

L'implantation des événements est générée automatiquement à partir des interfaces définies par le développeur. Cette approche me permet d'utiliser plus facilement des viviers d'objets et réduit le nombre d'objets créés.

Dans le modèle présenté, un stage ayant un ou plusieurs successeurs va utiliser son contexte afin d'envoyer des événements en sortie en utilisant la fonction dispatchToSuccessor(...). L'implantation du contexte est basée sur le patron de conception *décorateur* [37] pour ajouter les communications dans un stage. Pour connaître la politique de transmission vers les successeurs j'utilise des annotations Java spécifiées par le développeur en adéquation avec les types classiques de stages pris en considération par Saburo (*initial*, *final*, *default*, *collecteur*, *combineur*, *routeur* and *multicasteur*).

Le contexte du stage est généré automatiquement par GenContext selon le modèle de concurrence en utilisant les règles suivantes :

- si un seul processus est utilisé pour le traitement d'une requête, les communications sont réalisées sous forme d'appels de fonctions ;
- si plusieurs processus sont utilisés, les communications sont réalisées par des files locales et bloquantes ;
- pour des applications distribuées, les communications sont réalisées *via* des connexions réseaux.

Par exemple, le contexte du stage *read* pour le modèle de concurrence SPED est :

```
public class ReadStageContext extends StageContext< GeneratedEvent > {
  @Override
  public final void dispatchToSuccessor(GeneratedEvent event) {
    successor.handle(event) ;
  }
}
```

Dans le cas d'un serveur multi processus légers, le contexte du stage d'acceptation sera :

```
public class AcceptStageContext extends StageContext< GeneratedEvent > {
  @Override
  public final void dispatchToSuccessor(final GeneratedEvent event) {
    new Thread(new Runnable() {
        public void run() {
          while(successor.handle(event)) ;
    }).start() ;
}
```

La boucle principale du serveur est elle aussi générée automatiquement selon le modèle de concurrence spécifié par le développeur. Cette génération est réalisée par GenMain. Ainsi, si le modèle de concurrence nécessite des E/S non bloquantes, la boucle principale utilise le mécanisme de selection fourni par le langage Java afin de déterminer quels traitements doivent être réalisés. Dans Saburo, j'utilise la classe SaburoSelector pour réaliser cette selection. Cette classe d'encapsulation permet de réaliser une sélection sur une file locale et sur des interfaces d'E/S.

La boucle principale pour le modèle de concurrence *SPED* est :

```
while(true) {
  selector.doSelect();
}
```

Dans un modèle de concurrence utilisant des E/S bloquantes, il est juste nécessaire d'appeler la fonction handle(...) du stage accept. Ainsi pour le modèle de concurrence *itératif* la boucle principale est :

```
while(true) {
  acceptWrapper.handle();
}
```

Certains modèles de concurrence utilisent plusieurs processus légers. Dans ce cas, les deux boucles ci-dessus peuvent être réalisées dans des processus légers dédiés. Ainsi pour le modèle de concurrence *pipeline* on aura ce type d'instructions :

```
new Thread(new Runnable() {
  public void run() {
    while(true) {
      readWrapper.handle();
    }
  }
}
```

3.3.3 Problématique du décodage des requêtes clientes

Pour décoder les requêtes clientes, je vais devoir utiliser un analyseur lexical. Cet analyseur va me permettre de convertir le flot des caractères, en chaînes de caractères. La première étape est de spécifier les différents automates qui permettent de reconnaître les mots possibles du langage, ici les mots utilisables dans une requête HTTP. Par exemple, pour reconnaître le mot clé GET l'automate est le suivant :

```
public class DFA_Get implements DFA {
  private static int INIT = 0;
  private static int AFTER_G = 1;
  private static int AFTER_E = 2;
  private static int AFTER_T = 3;

  private int state = INIT;
```

```
  public DFAGet() {
    super();
  }

  public int step(char a) {
    switch (a) {
    case 'G' :
      if (state == INIT) {
        state = AFTER_G;
        return CONTINUE;
      } else {
        return REJECT;
      }
    case 'E' :
      if (state == AFTER_G) {
        state = AFTER_E;
        return CONTINUE;
      } else {
        return REJECT;
      }
    case 'T' :
      if(state == AFTER_E) {
        return ACCEPT;
      } else {
        return REJECT;
      }
    default :
      return REJECT;
    }
  }

  public void reset() {
    state = INIT;
  }
}
```

L'analyseur lexical va ainsi avoir l'ensemble des automates qui permettent de reconnaître tous les mots d'une requête HTTP. Le processus d'analyse va être le suivant :

```
private void processInput() {
  int currentPosition = startPosition;

  /*
  * Tant que l'on a encore au moins un automate qui est actif (c'est
  * à dire qui peut encore reconnaitre des caractères et que l'on n'est
  * pas en fin de chaine
  */
  while(currentPosition < buffer.length && activeDFA()) {
    /* On fait avancer l'ensemble de nos automates en parallèle! */
    for(int i = 0; i < dfa.length; ++i) {
      if(active[i]) {
        switch(dfa[i].step(buffer[currentPosition])) {
          /*
```

```
          * Si on a un rejet, l'automate n'est plus actif
          * et on a sauvegardé potentiellement des postions
          * intermèdiaires d'acception auparavant
          */
          case DFA.REJECT :
            active[i] = false;
            break;
          /*
          * Si on a une acceptation final, l'automate n'est plus actif
          * et on sauvegarde la position d'acceptation
          */
          case DFA.FINAL_ACCEPT :
            active[i] = false;
          /*
          * Si on a une simple acceptation, alors on sauvegarde la position
          * d'acceptation qui peut être intermèdiaire on reconnait quelque
          * chose d'autre par la suite ou final si on ne reconnait plus rien
          * par la suite
          */
          case DFA.ACCEPT :
            acceptPosition[i] = currentPosition + 1;
            break;
          }
        }
      }
    ++currentPosition;
  }
}
```

Ce processus d'analyse va « faire avancer » en parallèle des compteurs, i.e. des états, sur chacun des automates de reconnaissance. Lorsque l'on atteint la fin du mot (un espace) on va alors sélectionner l'automate qui reconnaît la plus longue chaîne. Deux remarques sont à effectuer :

1. les automates peuvent être produits automatiquement ;
2. seuls les automates changent en fonction du langage à reconnaître.

De plus, dans le cas non bloquant, le processus d'analyse peut s'interrompre à tout moment, car le flot de caractères n'est pas continu. Il est donc nécessaire de sauvegarder l'état du processus d'analyse, c'est-à-dire les positions courantes de chaque automate de reconnaissance. En effet, le processus qui exécute la reconnaissance lexicale va réaliser une autre tâche quand le flot de caractères sera interrompu. Lorsque de nouveaux caractères seront disponibles, il sera alors nécessaire de réinitialiser l'état de la tâche de reconnaissance. Cette sauvegarde d'état n'est pas à effectuer dans le cas bloquant.

Remarque : Les automates et les analyseurs sont « embarqués » dans des serveurs Internet. Les serveurs Internet sont des applications à durée de vie très longue et dont la gestion mémoire doit être fine. Pour économiser de l'espace mémoire, il peut n'y avoir qu'un seul ensemble d'automates de reconnaissance pour tous les analyseurs.

3.4 En conclusion...

Le développement d'applications repose encore aujourd'hui sur la construction de systèmes monolithiques qui doivent être maintenus et adaptés pour chaque évolution même s'il s'agit de développer des applications similaires. C'est pourquoi, le développement par « composition de composants et de services » a été proposé afin de répondre à ce problème de rationnalisation du développement d'applications. Il autorise une décomposition des fonctionnalités en entités plus petites et si possibles indépendantes.

J'ai présenté dans ce chapitre un outil dédié aux serveurs Internet, qui offre une bonne illustration de ce nouveau style de programmation, les *fabriques ou usines d'applications* [89]. Cet outil est assimilable à une chaîne de production et est basé sur trois concepts :

1. un graphe orienté de synchronisation et de communication ;
2. le code fonctionnel du serveur Internet ;
3. et, un modèle de concurrence choisi par le développeur.

A partir de ces trois concepts, je suis capable de composer automatiquement le serveur Internet fonctionnel. De plus, j'utilise à différents niveaux de ma fabrique de serveurs des concepts similaires à (i) la programmation par séparation des préoccupations basée sur la notion de programmation générative [30] et (ii) une programmation par « composants » simple et indépendante d'une technologie à composants particulière.

3.4.1 Vérification automatique à l'aide de *model checkers*

L'approche que je propose permet une meilleure sûreté des logiciels produits. En effet, elle permet d'éviter un grand nombre d'erreurs car le code concurrent traditionnellement sujet à de nombreux « comportements non désirés », est produit automatiquement. Elle offre aussi une facilité de portabilité et une prise en compte rapide de futures évolutions technologiques par simple ajout ou modification des règles de transformations du modèle de spécification vers les serveurs Internet produits. Enfin, elle offre une diminution du temps et des coûts de développement, permet aux développeurs de se consacrer à d'autres fonctionnalités du serveur Internet et nécessite moins de connaissances techniques.

Lors du processus de composition des serveurs Internet, j'ai constaté un important besoin de contrôle et de validation des assemblages. Ainsi, le graphe orienté de synchronisation et de communication y répond partiellement de « manière déclarative » pour les assemblages prévus tandis que les interfaces de contrôles de mes générateurs tentent d'y répondre de

« manière programmative ». La spécification sous forme de graphe orienté est particulièrement bien adaptée à l'application d'algèbres de processus, à une transformation vers un réseau de petri ou à la transformation vers un langage d'entrée d'un *model checker*. L'utilisation de tels outils permettra de renforcer de manière très significative la sûreté des serveurs Internet produit à l'aide de Saburo. Cependant plus la richesse du graphe, la complexité des différents stages et des données échangées seront importantes plus cette démarche apparaît comme difficile !

Est-il possible de simplifier la richesse des informations que l'on souhaite prendre en compte pour une vérification du serveur Internet et ainsi répondre aux besoins de validation des assemblages et de vérification des serveurs Internet développés à l'aide de Saburo ?

3.4.2 Décodage des requêtes clientes

J'ai illustré l'utilisation de Saburo, un prototype Java de mon outil de développement, en détaillant le développement d'un serveur HTTP. J'ai montré que l'étape du décodage des requêtes clientes nécessitent de prendre en compte le type d'E/S utilisé par le modèle de concurrence. En effet, si les E/S sont non bloquantes, il est alors nécessaire de sauvegarder le contexte puis de le restaurer à chaque fois. Ce qui n'est pas le cas pour des E/S bloquantes. Comme la principale motivation de Saburo est de permettre de passer très facilement et de manière transparente d'un modèle à un autre, il est donc nécessaire de proposer un outil permettant de s'abstraire du type des E/S.

Quel type d'outil peut on proposer pour s'abstraire du type des E/S et peut-on automatiser le développement de cette tâche longue, fastidueuse et peu difficile techniquement ?

Deuxième partie

Amélioration de la sûreté des serveurs Internet

4.

Vérification automatique de serveurs Internet

Sommaire

De nombreuses méthodes formelles ont été développées pour permettre la vérification systématique, et quelque fois automatique, d'un modèle formel et ainsi prouver que le système qui respecte ce dernier vérifiera certaines propriétés. La validité du modèle envisagé pour un système est alors assurée dès le début du développement. Il est largement accepté que cette approche permet de produire des logiciels plus sûrs. Elle est maintenant souvent utilisée pour la production d'applications dans les domaines sensibles.

Il existe actuellement deux grandes méthodes de vérification de logiciels. La première méthode est la *preuve de théorème* qui spécifie le système et les propriétés à vérifier sous forme de formules d'une logique mathématique. Le système va satisfaire une propriété si une preuve peut être construite depuis les axiomes du système pour cette propriété. Cette méthode est extrêmement puissante et elle est utilisée pour la vérification de système critique (vérification du logiciel de gestion du cœur d'une centrale nucléaire). Cependant, elle nécessite une grande expertise et est difficilement automatisable. La seconde méthode est le *model checking* qui permet de vérifier automatiquement des propriétés particulières d'un système exprimé sous forme de machine à états finie. Les propriétés à vérifier sont exprimées, elles aussi, sous forme de machine à états finie et peuvent être spécifiées sous forme de formules de logique temporelle. En *model checking*, la vérification consiste à explorer exhaustivement l'ensemble de l'espace d'états du système. En général, il est nécessaire d'en fournir une abstraction afin de limiter le nombre d'états à parcourir et ainsi éviter l'explosion du nombre d'états. Cependant, l'utilisation de ces méthodes formelles comme les réseaux de Petri [19, 93, 94, 96], les grammaires formelles [48], les algèbres de processus [46, 79, 80] ou

la logique temporelle [25, 61, 74] est généralement considérée comme difficile par la plupart des développeurs, ce qui nuit à leur utilisation. Elles deviennent difficiles à utiliser lorsque la complexité du système augmente. Leur rôle est donc souvent cantonné à la vérification de petits systèmes ou à une partie de ceux-ci.

Je pense que pour faciliter la diffusion des méthodes de vérification, il est nécessaire de proposer des formalismes simples et spécifiques au domaine d'application.

En effet, certaines propriétés ou caractéristiques dans un système d'un domaine donné, par exemple les interfaces graphiques, vont être différentes des propriétés ou des caractéristiques d'un système d'un autre domaine, comme les serveurs Internet. Par opposition aux formalismes introduits par les différentes méthodes de vérification, la représentation sous forme de graphe de mon modèle de développement est considérée comme un formalisme simple et largement répandu. De plus, les progès effectués par les outils de vérification automatique tel que le *model checking* permet de simplifier énormément la vérification et rend possible la vérification de certaines propriétés nativement. En effet, l'idée générale du *model checking* est de représenter un système sous forme d'une machine à états finie représentant tous les comportements possibles de ce système. La phase de vérification consiste à parcourir exhaustivement cette machine à états finie et à déterminer l'atteignabilité de tous les états et l'absence d'interbocage, c'est-à-dire que tous les états non-terminaux possèdent au moins un successeur.

En général, il est nécessaire d'écrire une abstraction du programme pour qu'il soit vérifiable par un *model checker*. En effet, le nombre d'états d'un programme, même simple, est souvent trop grand pour les *model checkers* actuels car il est nécessaire de modéliser l'ensemble des valeurs potentielles que peut prendre une donnée. Ecrire une bonne abstraction est une tâche importante car il faut réaliser un compromis entre le nombre d'états dans l'abstraction et l'intérêt des résultats obtenus lors de la vérification de l'abstraction. En particulier, l'abstraction doit être *sûre* vis-à-vis des propriétés à vérifier. C'est-à-dire que la présence d'une erreur dans le programme original doit être également trouvée dans l'abstraction.

Saburo utilise une approche descendante, c'est-à-dire que le modèle est généré à partir d'une meta-spécification. Ainsi, le modèle d'extraction que j'ai utilisé se base sur le graphe de spécification d'un serveur Internet ainsi que les modèles de concurrence comme langage d'entrée. Le code des différents stages n'est pas nécessaire pour obtenir l'abstraction du serveur Internet car l'ensemble des opérations de synchronisation et de communication présent dans un serveur Internet est modélisé *via* ce graphe de spécification. Ces informations vont être suffisantes pour la vérification des propriétés principales d'un serveur Internet que sont (i) l'absence d'interblocage et (ii) l'atteignabilité de tous les états du système. Il est très important de noter que l'abstraction et le serveur Internet fonctionnel sont obtenus automatiquement à partir d'une seule et unique spécification. D'autres travaux [43, 58, 82] utilisent une approche descendante mais ils ne se restreignent pas à un domaine particulier. Parmi ceux-ci, le projet HUGO [59] transforme une description UML en abstraction vérifiable à

l'aide d'un *model checker* ou d'une preuve par théorème et génére le code de l'application. Le modèle UML est vu comme une machine à états qui contient des classes actives (les états), des transitions, des interactions et d'autres contraintes. Les travaux de Sakharov [98] introduisent une spécification de machine à états finie en Java. Les transitions de cette machine sont étendues en ajoutant des expressions régulières sur les événements et en ajoutant des opérations de fusions sur les états par exemple.

D'autres outils sont quant à eux basés sur une approche ascendante, i.e. ils extraient le modèle formel à partir du code source. Ces approches peuvent être appliquées à n'importe quel programme et doivent généralement répondre au problème de l'explosion du nombre d'états dans la représentation formelle. Pour le langage Java, l'outil PathFinder [44] de la NASA est un *model checker* explicite qui analyse le bytecode des classes java directement pour détecter les interblocages et la violation d'assertion. Un autre outil, nommé Bandera [28] prend en entrée un code source Java et une spécification écrite dans le langage bien particulier de l'outil Bandera et va générer une spécification dans le langage d'entrée d'un *model checker* existant actuellement. Pour le langage C# ou C, Zing [4] est un *model checker* flexible passant bien à l'échelle pour les applications concurrentes. Il est conçu comme une seconde étape à un outil de génération de modèle qui extrait automatiquement des modèles comportementaux d'applications concurrentes.

Dans ce chapitre, je présente un exemple de génération automatique d'une abstraction sous forme d'une spécification en Promela. Promela est le langage de spécification du *model checker* SPIN qui est un des outil de *model checking* très utilisé. La spécification sous forme de graphe de mon modèle de développement est bien adaptée à la vérification et se traduit très simplement en Promela. Actuellement, mon approche supporte la vérification native, car fournie par le *model checker* SPIN, des propriétés comme l'interblocage ou l'atteignabilité. Il est possible d'ajouter des formules de logique temporelle pour améliorer les résultats de la vérification. Par exemple, pour vérifier que tous les clients recoivent une réponse valide ou une erreur du serveur.

4.1 Vérification de logiciels

Un premier objectif de la vérification de logiciels est de s'assurer que la spécification sait traiter toutes les combinaisons d'états et d'événements qui peuvent se produire. Plus précisemment, étant donné un état du modèle, il doit exister une règle pour traiter tous les événements susceptibles d'arriver dans cet état. Si une règle manque, comme dans le cas où un modèle envoie un message qui n'est jamais consommé, alors le concepteur a oublié une situation. L'effet d'une telle erreur peut être un interblocage. *La détection d'interblocage est l'un des objectifs principaux de la vérification.*

Un autre but de la vérification est de *s'assurer que la spécification ne contient pas d'état qui ne soit pas accessible.* De tels états reflètent en général des erreurs de conceptions, comme un état qui peut être atteint uniquement suite à la réception d'un événement qui n'arrive jamais.

Une spécification qui ne contient pas d'interblocage et dont tous les états sont accessibles peut encore contenir des erreurs logiques, c'est-à-dire qu'elle se comporte différemment de ce à quoi l'on s'attend d'elle. Par exemple dans le cas d'une session d'authenfication, la fonction sémantique de vérification du mot de passe répond toujours positivement.

Mon modèle de développement a été conçu de telle sorte qu'une abstraction puisse être facilement déduite de la spécification originale. L'idée principale de cette abstraction est de supprimer toute la partie sémantique des différents stages, sous réserve qu'elle ne contient aucun code de synchronisation, et de ne conserver que la partie communication et synchronisation exprimée par mon graphe de spécification et le modèle de concurrence. Ceci introduit nécessairement une certaine imprécision dans la description du comportement d'une application, permet de restreindre le nombre d'états du système et permet la vérification des propriétés d'atteignabilité et d'absence d'interblocage qui sont prédominantes dans un serveur Internet.

4.2 Génération automatique d'un modèle formel

Dans cette section, je vais construire progressivement la traduction d'une spécification d'un serveur Internet en langage de spécification d'un *model checker*. La traduction ainsi obtenue sera utilisée pour traduire dans mon cas d'étude (section 4.3) la spécification d'un serveur Internet vers le langage Promela, langage d'entrée du *model checker* SPIN. La représentation, sous forme d'un graphe orienté, que j'ai utilisé pour décrire un serveur Internet est particulièrement bien adaptée à sa traduction vers une spécification d'un *model checker*. Je vais présenter dans la section 4.3, un exemple concret illustrant cette traduction vers le langage Promela.

4.2.1 Transducteur et transducteur abstrait

Dans le but de définir précisemment la traduction d'une spécification d'un serveur Internet en langage de spécification d'un *model checker*, une spécification de mon modèle de développement est vue comme un transducteur $T_{(G,C)}$ qui traduit une suite d'événements en entrée en une suite d'événements en sortie. Les événements en sortie vont dépendre du graphe de spécification G et du modèle de concurrence C.

Les règles strictes de spécification que j'ai imposées dans mon modèle de développement permettent d'obtenir facilement une abstraction d'un système en éliminant la partie sémantique des stages tout en conservant les propriétés d'atteignabilité et de concurrence d'un serveur Internet. En effet, le graphe de spécification G et son transducteur associé $T_{(G,C)}$ correspondent, par construction, à l'énumération de toutes les synchronisations et communications entre stages qu'un serveur Internet réalisent pour fournir un service. Je vais noter $Abst(T_{(G,C)})$ le transducteur obtenu en supprimant la partie sémantique des différents stages de $T_{(G,C)}$.

4.2.2 Atteignabilité et interblocage

Un état S' est un *successeur* d'un état S si S' peut être atteint depuis S par une unique transition. Un état S' est *atteignable* depuis S s'il est un successeur de S ou s'il est un successeur d'un état accessible depuis S. Chaque transducteur a un ensemble d'états initiaux. On définit l'*atteignabilité* $Reach(T_{(G,C)})$ d'un transducteur $T_{(G,C)}$ comme l'ensemble des états atteignables depuis les états initiaux. Par construction de $Abst(T_{(G,C)})$, il est facile de voir :

$$Abst(Reach(T_{(G,C)})) \subseteq Reach(Abst(T_{(G,C)})) \quad (1)$$

c'est-à-dire que l'abstraction des états accessibles de $T_{(G,C)}$ est un sous-ensemble des états accessibles de l'abstraction de $T_{(G,C)}$.

Etant donné un transducteur $T_{(G,C)}$, si l'on suppose que toutes les actions sémantiques des différents stages se terminent quelles que soient leur valeurs d'entrée et leur contexte, un *interblocage* pour $T_{(G,C)}$ est un état non terminal de $T_{(G,C)}$ qui n'a pas de successeur. Cette propriété est vérifiée car dans mon modèle de développement, j'impose que les stages n'implantent pas d'opérations de synchronisation sur des informations échangées entre stages. En effet, c'est le graphe de spécification qui modélise toutes les synchronisations et communications entre les stages !

L'abstraction du transducteur $T_{(G,C)}$ obtenue par la fonction $Abst$ est valide pour la détection des interblocages et l'analyse des états accessibles. Plus précisémment, les propriétés suivantes sont la conséquence de la propriété 1.

Propriété 1 *Atteignabilité et absence d'interblocage :*

1. *pour chaque état s_i de $T_{(G,C)}$, si $Abst(s_i)$ n'est pas atteignable dans $Abst(T_{(G,C)})$ alors s_i n'est pas atteignable dans $T_{(G,C)}$;*
2. *si $Abst(T_{(G,C)})$ ne contient pas d'interblocage, alors $T_{(G,C)}$ ne contient pas d'interblocage.*

Ces propriétés montrent que l'abstraction proposée est sûre. Toutefois, la vérification sur la version abstraite peut trouver des *faux positifs*, c'est-à-dire des erreurs qui n'existent que dans l'abstraction. Plus généralement, les inverses des propriétés précédentes peuvent ne pas être vraies. Par exemple, dans le cas d'un stage de type routeur, une branche peut ne pas être accessible car le prédicat de routage est toujours faux. Par contre, lors de la transduction, les prédicats de routage sont généralement remplacés par de simples alternatives non déterministes pour limiter l'espace de recherche, ce qui contredit la première propriété.

4.2.3 Transduction finie et infinie

Souvent, le transducteur $Abst(T_{(G,C)})$ est fini et l'abstraction peut alors être vérifiée avec les méthodes conventionnelles de *model checking*. De plus, pour les serveurs Internet, le nombre d'états du transducteur $Abst(T_{(G,C)})$ est en général assez faible et très en-dessous des limites des outils de *model checking*. Par exemple, le transducteur correspondant à la

modélisation du serveur HTTP a quelques dizaines d'états. Exceptionnellement, ce nombre peut dépasser la vingtaine d'états et les *model checkers* actuels travaillent généralement sur plus d'une centaine d'états.

Cependant, pour certains modèles de concurrence, le transducteur $Abst(T_{(G,C)})$ n'est pas fini. En effet, le nombre de processus légers créés par le modèle n'est pas toujours borné. Par exemple, si le développeur d'un serveur Internet veut créer autant de processus légers qu'il le désire et donc qu'il n'utilise pas de vivier de processus légers. Dans ce cas, le transducteur $Abst(T_{(G,C)})$ est non borné et ne peut pas être vérifié par les techniques classiques de *model checking* !

Afin de vérifier des systèmes non bornés, je vais introduire une nouvelle approximation aux spécifications de serveurs Internet en fixant une borne supérieure N au nombre de processus légers qui peuvent être créés. J'obtiens ainsi une approximation finie en nombre d'états du transducteur, notée $Abst_N(T_{(G,C)})$ et ce transducteur peut être vérifié par les *model checkers* classique, si la borne N n'est pas trop grande. Un grand nombre de résultats de vérification obtenus sur $Abst_N(T_{(G,C)})$ peuvent être généralisés à $Abst(T_{(G,C)})$, puis à $T_{(G,C)}$. En particulier, les propriétés suivantes sont vraies :

Propriété 2 *Transducteur borné, atteignabilité et absence d'interblocage :*
Soit $Abst(T_{(G,C)})$ un transducteur non borné.

1. *si un état s_i est atteignable dans $Abst_N(T_{(G,C)})$, alors il est atteignable dans $Abst(T_{(G,C)})$;*
2. *s'il y a un interblocage dans $Abst_N(T_{(G,C)})$, alors il y a aussi un interblocage dans $Abst(T_{(G,C)})$.*

Cependant, ces propriétés ne donnent aucune garantie sur la généralisation des résultats de vérification obtenus sur l'abstraction $Abst_N(T_{(G,C)})$, qui a un nombre fini d'états, pour la version non bornée $Abst(T_{(G,C)})$. $Abst_N(T_{(G,C)})$ est donc une abstraction non sûre de $Abst(T_{(G,C)})$. Par exemple, si un interblocage apparaît dans $Abst(T_{(G,C)})$ lorsque le nombre de processus légers créés est supérieur à $k > 0$, alors il ne sera pas possible de détecter cet interblocage dans $Abst_N(T_{(G,C)})$ si $N < k$.

Dans la pratique, soit le système n'est pas valide pour des petites valeurs de N, soit il est valide quelle que soit la valeur de N. Ainsi, même pour de petites valeurs de N, il est possible d'augmenter la confiance dans la spécification. De plus, cela permet éventuellement de détecter des erreurs dans la spécification même si les résultats ne peuvent pas être généralisés automatiquement.

Cette transduction finie, obtenue par raffinement, va être utilisée pour traduire ma spécification de serveurs Internet vers le langage Promela, langage d'entrée de SPIN.

4.3 Un exemple concret, traduction vers le langage Promela

Les notations et les résultats présentés dans les sections précédentes ont été implantés dans Saburo, mon outil de développement de serveurs Internet en Java. Ce prototype dé-

veloppé en Java, est un « compilateur » prenant en entrée une spécification d'un serveur Internet sous forme de graphe et un modèle de concurrence afin de générer en sortie un modèle formel en Promela, langage de spécification du *model checker* SPIN [47]. Comme tout compilateur, il est possible de changer la partie de génération de code, afin de produire une spécification pour d'autres *model checkers*, tel que NuSMV [24] ou TSMV [75].

4.3.1 Le langage Promela

Le langage Promela (*PROtocol MEta LAnguage*) est un langage de spécification de systèmes asynchrones. Ce qui en d'autres termes veut dire que ce langage permet la description de systèmes concurrents, comme les protocoles de communication. Il autorise la création dynamique de processus. La communication entre ces différents processus peut se faire en partageant les variables globales ou alors en utilisant des canaux de communication. On peut ainsi simuler des communications synchrones ou asynchrones. En Promela, il n'y a pas de différence entre les instructions et les conditions. Une instruction ne peut être passée que si elle est exécutable, une condition que si elle est vraie. Sinon le processus est bloqué jusqu'à ce que la condition devienne vraie.

Promela est un langage impératif qui ressemble au langage C agrémenté de quelques primitives de communication. Pour communiquer, les processus peuvent utiliser des canaux de communication *FIFO*, prendre rendez-vous ou utiliser des variables partagées. Un programme Promela est une liste de déclarations de processus, de canaux et des variables.

Déclaration de variables

On indique le type (bit, byte, short ou int), le nom de la variable et optionnellement sa valeur initiale.

```
bool b1 = false, b2 = false ; bit k = 0 ;
```

Les variables de type tableau sont déclarées comme en C, par exemple :

```
bit porteouverte[3] ;
```

Déclaration de canaux

On l'introduit par le mot clé `chan`, suivi du nom du canal et optionnellement de la longueur du FIFO et du type des messages qui circulent. Par exemple :

```
chan Ouvreporte = [0] of { byte, bit } ;
chan Transfert = [2] of { bit, short, chan } ;
```

Ouvreporte est un canal synchrone, car sa longueur est 0, ce qui correspond à un rendez-vous. Sur ce canal circulent des messages ayant une partie byte et une partie bit. Transfert est un canal asynchrone, car il peut stocker (au plus) deux messages. Sur ce canal circulent des messages composés de trois parties. La première bit, la seconde de type short et la dernière de type chan.

Déclaration de processus

La forme la plus simple de déclaration de processus est :

```
proctype nom ( paramètres_formels ) {
  instructions
}
```

Un processus est instancié en utilisant l'instruction *run* :

```
run nom ( paramètres )
```

Ainsi, par exemple :

```
proctype P (bit i) {
  ...
}

proctype porte (byte i) {   ...
}

proctype ascenseur () {
  ...
}
```

Opérations sur les canaux

Sur un canal on peut envoyer (opération *!*) ou recevoir (opération *?*) des messages. Par exemple :

```
ouvreporte !i,0 ;
ouvreporte ?i,1 ;
ouvreporte ?eval(etage),1
```

La fonction *eval* force l'égalité des valeurs reçues avec etage, la variable *etage* n'est pas changée.

Expressions

Une expression peut être utilisée comme une instruction si elle ne fait pas des effets de bord (opérations "–" et "++" de C). Dans ce cas, elle est exécutable quand sa valeur devient vraie (par le changement des valeurs des variables partagées). Par exemple :

```
(a == b) ;
```

est équivalent à :

```
while (a != b) skip ;
```

Instruction *init*

L'exécution du système commence par le processus *init* (c'est la fonction *main* de Promela). Par exemple :

```
init {
  run porte(1) ;
  run porte(2) ;
  run porte(3) ;
  run ascenseur() ;
}
```

Instruction *atomic*

L'exécution d'une séquence d'instructions préfixée par *atomic* est rendue indivisible, c'est-à-dire sans l'entrelacement des actions des autres processus. Par exemple :

```
atomic {
  run porte(1) ; run porte(2) ; run porte(3) ;
  run ascenseur() ;
}
```

Instruction *if*

Une branche de l'instruction *if* est exécutable si la première instruction de la branche, appelée aussi garde, est vraie. L'instruction *if* bloque jusqu'à ce qu'une branche devienne exécutable. Si plusieurs branches sont exécutables, alors l'une d'entre elles est choisie arbitrairement (non-déterminisme).

L'exemple suivant incrémente ou décrémente la valeur de *count* une fois.

```
if
: : count = count + 1
: : count = count - 1
fi
```

Instruction *do*

Cette instruction est similaire à l'instruction *if*, sauf que l'instruction est exécutée et puis la sélection est répétée jusqu'à l'exécution d'une instruction *break*.

```
proctype ascenseur () {
  show byte etage = 1 ;

  do
  : : (etage != 3) ->
    etage++ ;
  : : (etage != 1) ->
    etage-- ;
  : : ouvreporte !etage,1 ;
    ouvreporte ?eval(etage),0 ;
  od
}
```

4.3.2 Génération automatique d'un modèle de serveur Internet en Promela

Promela modélise une application en utilisant le principe de système événementiel. Le graphe qui modélise les communications et les synchronisations entre les stages d'un serveur Internet est un système événementiel. Les communications entre le serveur et les clients font aussi partie d'un système événementiel.

4.3.2.1 Modélisation des événements

Les évéments utilisés pour la communication entre les stages ainsi que pour la communication avec les clients sont modélisés en utilisant des valeurs symboliques. Ceci permet de s'abstraire d'une valeur spécifique et rend les noms des constantes disponibles à l'implantation, et permet un meilleur retour d'erreur pour le développeur.

Par exemple pour le serveur Internet on aura les événements suivants :

```
mtype = REQUEST, RESPONSE, CLOSE, NULL;
mtype = OP_ACCEPT, OK_ACCEPT, NO_ACCEPT;
mtype = OP_READ, OK_READ, NO_READ;
mtype = OP_WRITE, OK_WRITE, NO_WRITE;
...
```

La convention de nommage respecte la règle suivante :

<type>_<operation>

Avec, type qui peut prendre OP, OK ou bien NO comme valeur. Ces valeurs représentent respectivement si le système demande une action ou reçoit une réponse de celui-ci. operation correspond à l'action réalisée par le stage.

4.3.2.2 Modélisation de la socket serveur

La socket serveur qui attend les connexions clientes est modélisée comme un canal qui peut stocker MAX_MESSAGE messages. En fonction du modèle de concurrence, la taille du tampon est fixé à 0 afin de forcer la synchronisation durant l'échange de message ou possède une taille supérieure à 0 pour permettre l'échange de message asynchrone.

4.3.2.3 Modélisation des sockets clientes

La socket cliente, le point de communication entre un client et le serveur est modélisée à l'aide de deux canaux de communication. En effet, le client et le serveur ont le même comportement de producteur et de consommateur d'événements. Il est donc nécessaire d'utiliser

deux canaux. Le serveur va lire dans l'un des deux canaux et écrire dans l'autre. Le client aura le comportement inverse, i.e. il écrira dans le premier canal et lira dans le second.

```
chan s_read;
chan s_write;
```

Le premier canal s_read est envoyé par le client au serveur au moment de la connexion. Le serveur va l'utiliser pour lire des événements provenant du client. Le second canal s_write est envoyé par le serveur au client au moment de l'acceptation de celui-ci. Le serveur l'utilise pour envoyer des événements au client.

4.3.2.4 Modélisation des E/S

Afin de modéliser les connexions des clients et les erreurs potentielles du serveur ou des clients, les communications sont modélisées à l'aide de traitement non déterministe. L'ajout de ces traitements non déterministes dans une spécification augmente l'espace du nombre d'états à explorer par le *model checker*. Cette augmentation de la mémoire peut être de l'ordre d'un facteur deux [47]. Cependant, grâce à la méthode d'obtention de l'abstraction, cette augmentation mémoire reste acceptable.

Par exemple, pour modéliser les lectures et les erreurs de lecture, Saburo ajoute les traitements non déterministes suivants :

```
if
:: s_write!OK_READ( s_read ) ->
   ...
:: s_write!NO_READ( s_read ) ->
   ...
fi;
```

4.3.2.5 Modélisation du graphe des stages

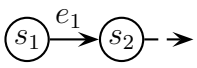
Précédemment, j'ai expliqué comment modéliser les événements, les canaux de communication et le modèle d'E/S. Je vais maintenant détailler comment le graphe de communication et de synchronisation de Saburo est transformé en instructions Promela.

A partir de la spécification des stages, Saburo transforme chaque stage en utilisant des patrons Promela correspondant à son type. Sept types de stages classiques [63] sont pris en compte par Saburo (*initial*, *final*, *default*, *collecteur*, *combineur*, *routeur* et *multicasteur*). Ces différents types de stage et les instructions Promela produites par la phase de génération

sont donnés dans les figures ci-dessous. Par la suite, je noterai par $c1$ le canal utilisé par le stage $s1$ pour communiquer avec son predecesseur.

Stage initial

Un stage *initial* n'a pas de prédecesseur et envoie un événement vers un seul successeur (voir Fig. 4.1).



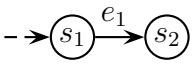

```
s1 : c2 !e1 ;
s2 : c2 ?e1 ;
```

FIG. 4.1 – Le stage s_1 est un stage inital.

Le stage s_1 ne peut recevoir d'événements, car c'est un stage initial. Il envoie l'événement e_1 au stage s_2 *via* le canal c_2.

Stage final

Un stage *final* n'a pas de successeur et reçoit un évévement d'un seul précédesseur (voir Fig. 4.2).




```
s1 : c1 ?e ;
     c2 !e ;
s2 : c2 ?e ;
```

FIG. 4.2 – Le stage s_2 est un stage final.

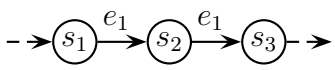
Le stage s_2 reçoit l'événement e_1 du stage s_1 *via* le canal c_2 et ne le retransmettra pas car c'est un stage final.

Stage défaut

Un stage *default* reçoit des évéments d'un seul prédecesseur et les relais vers un seul successeur (voir Fig. 4.3).

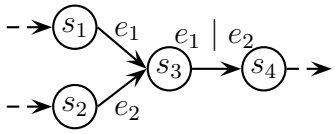
Le stage s_1 envoie l'événement e_1 au stage s_2 *via* le canal c_2. Le stage s_2 va bloquer tant qu'il n'aura pas reçu l'évément e_1. Puis il va le retransmettre au stage s_3 *via* le canal c_3.

Stage collecteur




```
s1 : c2 !e1 ;
s2 : c2 ?e1 ;
     c3 !e1 ;
s3 : c3 ?e1 ;
```

FIG. 4.3 – Le stage s_2 est un stage *défaut*.




```
s1 : c3 !e1 ;
s2 : c3 !e2 ;
s3 : c3 ?e ;
     c4 !e ;
s4 : c4 ?e ;
```

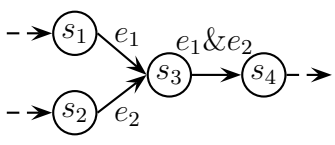
FIG. 4.4 – Le stage s_3 est un collecteur.

Un stage *collecteur* reçoit des événements depuis plusieurs précédesseurs et les relaie vers un seul successeur (voir Fig. 4.4).

Le stage s_1 (respectivement s_2) envoie l'événement e_1 (respectivement e_2) au stage s_3 *via* le canal c_3. Le stage s_3 transmet l'événement en entrée à son successeur s_4 *via* le canal c_4.

Stage combineur

Un stage *combineur* va bloquer tant qu'il ne peut pas combiner un évément provenant de chacun de ses prédecesseurs et les relaie vers un seul successeur (voir Fig. 4.5).



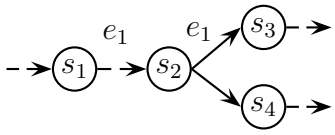

```
s1 : c3 !e1 ;
s2 : c3 !e2 ;
s3 :
     if
     : : c3 ?e1 ->
         c3 ?e2 ;
         c4 !(e1, e2) ;
     : : c3 ?e2 ->
         c3 ?e1 ;
         c4 !(e1, e2) ;
     fi ;
s4 : c4 ?(e1, e2) ;
```

FIG. 4.5 – Le stage s_3 est un combineur.

Le stage s_1 (respectivement s_2) envoie l'événement e_1 (respectivement e_2) au stage s_3 *via* le canal c_3. Le stage s_3 bloque tant qu'il n'a pas reçu au moins un événement de chacun de ses prédecesseurs. Il transmet à son successeur s_4 une « combinaison » de ses événements en entrée *via* le canal c_4.

Stage routeur

Un stage *routeur* va aiguiller tous les événements obéissant à un prédicat vers un successeur donné et les autres vers un autre successeur (voir Fig. 4.6).




```
s1 : c2 !e1 ;
s2 : c2 ?e1 ;
   if
   : : c3 !e1 → . . . ;
   : : c4 !e1 → . . . ;
   fi ;
```

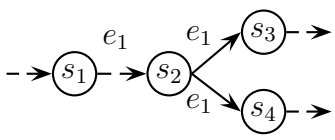
FIG. 4.6 – le stage s_2 est un routeur.

Le stage s_2 reçoit *via* le canal c_2 un événement e_1 de son prédecesseur s_1 et aiguille tous les événements obéissant à un prédicat donné vers son successeur s_3 *via* le canal c_3 et tous les autres vers son successeur s_4 *via* le canal c_4.

Remarque : Au lieu d'utiliser une construction de type if . . . else . . . fi, un stage routeur va avoir un comportement non déterministe grâce aux instructions Promela adéquates.

Stage multicasteur

Un stage *multicasteur* envoie le même événement à l'ensemble de ses successeurs (voir Fig. 4.7).




```
s1 : c2 !e1 ;
s2 : c2 ?e1 ;
   c3 !e1 ;
   c4 !e1 ;
```

FIG. 4.7 – Le stage s_2 est un multicasteur.

Le stage s_2 reçoit *via* le canal c_2 un événement e_1 de son prédecesseur s_1 et retransmet cet événement vers l'ensemble des ses successeurs.

Remarques : Comme nous l'avons vu précédemment, le type des stages est spécifié par le développeur en utilisant le mécanisme des annotations Java [18]. Le type est ensuite obtenu en utilisant les mécanismes de réflexion fournis par le langage Java [103]. Les connexions entre les stages sont déduites par le parcours du graphe spécifié par le développeur. Ce graphe est contenu dans la classe StageGraphManager de l'API de Saburo.

Les stages *Combineur* et *Routeur* nécessitent des annotations avec des arguments pour décrire leur comportement. Ainsi pour un *combineur*, une expression logique en Java est utilisée pour décrire comment les événements d'entrée doivent être combinés. L'annotation pour le stage *combineur* donné en exemple ci-dessus est :

```
@Combiner(E1 & E2)
```

Pour un stage *routeur*, un prédicat doit être utilisé pour aiguiller les événements d'entrée. L'annotation pour le stage *routeur* donné en exemple ci-dessus est :

```
@Router((E1,s3))
```

Les instructions E1 et E2 correspondent aux interfaces Java des événements e1 et e2. Dans l'expression logique d'un stage *routeur*, l'événement E1 est associé au stage s3 vers lequel il sera redirigé. Dans le cas d'un stage *combineur*, il n'est pas nécessaire de réaliser l'association stage-événement car la connexion entre les stages sont explicites dans le graphe de spécification des serveurs. Les événements sont associés à cette connexion *via* le prototype de la fonction handle(...).

4.3.2.6 Modélisation du modèle de concurrence

La génération des instructions Promela dépend du modèle de concurrence choisi par le développeur. Je vais maintenant détailler la transformation du serveur HTTP utilisant le modèle de concurrence SPED.

Comme un seul processus est utilisé dans ce modèle, seules les communications internes sont à spécifier. La communication directe des événements n'est pas possible sinon on se retrouverait à modéliser le modèle de concurrence itératif. Ainsi, pour permettre de mélanger les connexions et les événements d'E/S un canal unique est utilisé pour passer les événements. Il est unique car un seul processus est utilisé.

Pour modéliser l'entrelacement de plusieurs requêtes, un vivier de canaux est défini. Ces canaux vont être utilisés par le serveur pour modéliser la réception des événements des clients. Accepter une nouvelle connexion cliente revient ainsi à extraire un canal de ce vivier.

Le vivier va être modélisé de la façon suivante :

```
chan pool = [MAX_CLIENT] of { chan };
```

Pour extraire un canal de ce vivier, on détermine si le vivier est vide ou non en utilisant la fonction nempty(). Si tel est le cas, l'appel va bloquer tant qu'un canal n'est pas mis à disposition dans le vivier. Cependant si le vivier contient un canal, on va l'extraire en utilisant l'opération de réception d'un message du langage Promela. Ces opérations vont être modélisées de la façon suivante :

```
if
:: nempty( pool ) ->
   pool ?s_read ;
fi ;
```

Du point de vue du langage Promela, le modèle SPED est un modèle copératif utilisant un unique processus. La boucle principale du serveur est modélisée par une boucle infinie do ...od;. Tous les événements de communication internes sont envoyés et reçus dans l'unique canal. La boucle principale do ...od; loop est marquée comme étant un état terminal valide à l'aide du marqueur end.

```
end : do
      :: server ?< REQUEST, s_write > ->
      :: internal ?OP_ACCEPT( s_write, s_read ) ->
         ...
         internal !OP_READ( s_write, s_read ) ;
         ...
      :: internal ?OP_READ( s_write, s_read ) ->
         ...
         internal !OP_WRITE( s_write, s_read ) ;
         ...
      :: internal ?OP_WRITE( s_write, s_read ) ->
         ...
         s_write !RESPONSE( s_read ) ;
         s_read ?CLOSE( s_write ) ;
         goto release ;
      od ;
```

Le code complet de cet exemple de modélisation d'un serveur HTTP simple pour le modèle SPED, ainsi que celui de tous les autres modèles de concurrence pris en compte par Saburo, sont donnés dans l'annexe A.

4.3.3 Simulation et vérification en pratique

Le logiciel SPIN (*Simple Promela INterpreter*) est un outil utilisé pour simuler et vérifier un programme écrit en Promela. Le logiciel SPIN comporte essentiellement deux modes :

1. *simulation :* le système est exécuté pas à pas, ce qui permet de se familiariser avec son comportement.
2. *vérification :* les états du système sont explorés exhaustivement pour vérifier que le système satisfait bien certaines propriétés exprimées.

4.3.3.1 Vérification d'un système

Les instructions Promela obtenues de la transformation sont utilisées comme entrée au *model checker* SPIN. Cet outil permet de vérifier automatiquement que tous les états sont atteignables et l'absence d'interblocage de la spécification sans autre spécification. Les résultats de la vérification du modèle SPED du serveur HTTP sont donnés ci-dessous :

```
warning : for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
depth 0 : Claim reached state 5 (line 164)
(Spin Version 4.2.6 – 27 October 2005)
  + Partial Order Reduction

Full statespace search for :
  never claim +
  assertion violations + (if within scope of claim)
  acceptance cycles + (fairness disabled)
  invalid end states - (disabled by never claim)

State-vector 198 byte, depth reached 191, errors : 0
51946 states, stored
37936 states, matched
89882 transitions (= stored+matched)
  176 atomic steps
hash conflicts : 1043 (resolved)

Stats on memory usage (in Megabytes) :
10.701 equivalent memory usage for states (stored*(State-vector + overhead))
8.598 actual memory usage for states (compression : 80.35%)
  State-vector as stored = 158 byte + 8 byte overhead
2.097 memory used for hash table (-w19)
0.320 memory used for DFS stack (-m10000)
0.123 other (proc and chan stacks)
0.099 memory lost to fragmentation
10.916 total actual memory usage


unreached in proctype SpedHttp
  line 110, "pan._ _ _", state 67, "-end-"
  (1 of 67 states)
```

```
unreached in proctype Client
  (0 of 35 states)
unreached in proctype :never :
  line 169, "pan._ _ _", state 8, "-end-"
  (1 of 8 states)
0.76 real 0.70 user 0.03 sys
```

Le résultat présenté ci-dessus correspond à la vérification de l'absence d'interblocages et de l'atteignabilité de l'ensemble des états d'un serveur HTTP basé sur le modèle de concurrence SPED. On peut voir que le système est représenté par 51946 états reliés par 89882 transitions. La consommation mémoire est détaillée et environ 11 méga octets sont utilisés. On peut observer que les états marqués par le mot clef `end` sont bien les seuls états terminaux du système.

4.3.3.2 Simulation d'un système

SPIN permet aussi de simuler le comportement d'un système. La figure suivante montre la trace de la simulation de l'abstraction du serveur HTTP utilisant le modèle SPED. Cette trace est obtenue à l'aide de l'outils XSPIN. On peut voir que les événements sont entrelacés sur cette trace. En particulier les opérations de lecture et d'écriture de deux clients connectés sont entrelacées sur cette trace.

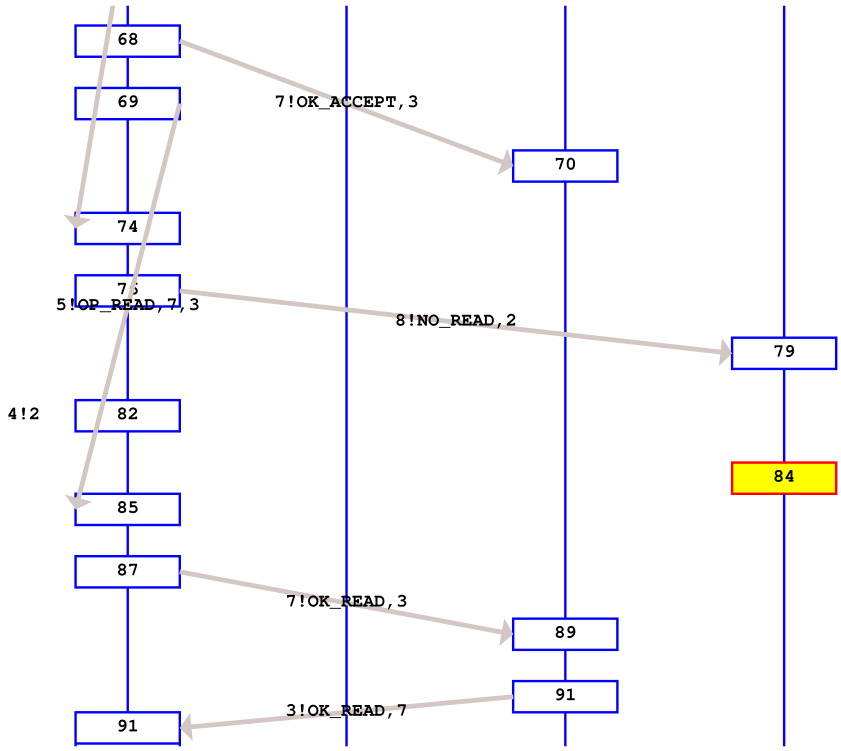


FIG. 4.8 – Une partie d'une simulation obtenue à l'aide de l'outil graphique XSPIN

Les numéros donnés sur la figure correspondent aux différents états du système. La flèche entre le numéro 68 et le numéro 70 indique que le serveur accepte le deuxième client. La communication du serveur au second client se fait *via* les canaux 7 et 3. Le canal 7 (respectivement 3) est utilisé par le serveur (respectivement le client) pour envoyer des messages au client 2 (respectivement serveur). On peut voir (message 5!OP_READ,7,3) que le serveur passe de l'état d'acceptation à l'état lecture pour le client 2. Le client 3 a atteint un état final valide (numéro 84, boîte jaune) car il a reçu du serveur le message l'informant d'une erreur de lecture.

4.3.4 Application de formules de logique temporelle linéraire

En plus des vérifications automatiques de l'atteignabilité de tous les états et de l'absence d'interblocage, il est possible de rajouter des formules de logique temporelle linéaire (*Linear Temporal Logic*. Ces formules peuvent être rajoutées par le développeur *via* l'interface SPIN. Ces formules permettent d'augmenter considérablement les bénéfices de la phase de vérification mais nécessite un plus gros effort dans l'apprentissage de cette logique temporelle et de l'outil SPIN.

En voici quelques exemples

La formule LTL suivante vérifie que si un client est connecté à un serveur, il reçoit obligatoirement soit (i) une réponse ou (ii) une erreur. Cette propriété de *vivacité* doit être vraie dans les états terminaux du système que l'on modélise. Ces états terminaux vont dépendre du modèle de concurrence utilisé.

$$\diamond(request == (response + error)) \tag{2}$$

Cet autre exemple de formule LTL vérifie que les clients peuvent toujours se connecter au serveur. Cette propriété de *sûreté* doit être vraie dans tous les états du système !

$$\square(len(serverSocket) <= MAX\_CLIENT) \tag{3}$$

La fonction len est une fonction prédéfinie du langage Promela. Cette fonction permet de déterminer le nombre de messages stockés dans un canal.

Ces formules sont ajoutées et vérifiées sur les instructions Promela obtenues automatiquement à l'aide de Saburo.

4.4 En conclusion...

L'augmentation de la sûreté d'une application est permise par l'utilisation d'outils de vérification automatique, appelés *model checkers*. Néanmoins, leur utilisation est perçue comme difficile. De plus, lors de la vérification automatique d'un système, ces méthodes vont réaliser une exploration exhaustive, i.e. combinatoire, de tous les états de la représentation formelle de celui-ci. Pour résoudre ce problème d'explosion du nombre d'états, il est nécessaire d'écrire

une abstraction d'un programme pour qu'il soit facilement vérifiable par un outil de vérification automatique.

L'écriture d'une bonne abstraction est importante et difficile car il est nécessaire de réaliser un compromis entre le nombre d'états dans l'abstraction et l'intérêt des résultats obtenus lors de la vérification. Ainsi la présence d'une erreur dans le programme original doit également être retrouvée dans l'abstraction (propriété de *sûreté*). Cependant, l'abstraction est bien souvent spécifiée à la main, ce qui peut introduire un biais entre le modèle et l'application qu'il modélise. Pour éviter ce biais, il est préférable que l'abstraction soit obtenue automatiquement à partir du système à vérifier. Il existe des logiciels générant automatiquement des abstractions sûres pour n'importe quel programme C ou Java [4, 28, 44]. Mais ces approches sont trop générales et nécessitent de spécifier formellement des règles d'extraction de l'abstraction en fonction des différentes propriétés que l'on souhaite vérifier. Il est évident que la spécification formelle de ces différentes règles d'extraction complexifie l'utilisation de tels outils.

C'est pourquoi j'ai proposé des générateurs d'abstractions qui sont spécifiques à un domaine précis, les serveurs Internet [71]. Se focaliser sur un seul domaine permet d'éviter de spécifier les propriétés que l'on souhaite vérifier car elles dépendent bien souvent du domaine, simplifie le développement des générateurs et améliore la sûreté des abstractions vis-à-vis du système. Ainsi, Saburo applique aux serveurs Internet des concepts similaires à ceux développés dans VEG [14].

Mon approche *descendante et spécifique* permet d'*obtenir automatiquement les abstractions* sans avoir à spécifier les propriétés à vérifier. Pour accroître la sûreté de l'application, il est possible d'*ajouter des propriétés supplémentaires* à vérifier, sous forme de formules de logiques temporelles. La sûreté de l'abstraction vis-à-vis de l'application qu'elle modélise est garantie par les différents générateurs que je fournis. Elle permet aussi de *simplifier considérablement le développement des générateurs d'abstractions* comparativement à ces méthodes [4, 28, 44]. En effet, le langage d'entrée, dans mon cas le graphe de spécification du serveur et le modèle de concurrence, est extrêmement simple comparativement à un langage de programmation complet tel que le C ou le Java.

Plus pratiquement, dans Saburo la phase de vérification est basée sur le *model checker* SPIN [47] et j'utilise ici aussi des générateurs pour transformer la spécification en un modèle de vérification en Promela. La spécification est indépendante de tout modèle de concurrence. Le modèle est choisi parmi un ensemble prédéfini à la phase de génération.

Il existe d'autres modèles checker qui pésentent des caractéristiques différentes de SPIN. Est-il possible d'incorporer à Saburo des générateurs de spécification pour d'autres modèles checker ?

5

Utilisation d'un générateur d'analyseurs syntaxiques pour l'implantation des protocoles

Sommaire

Dans un serveur Internet, le décodage des requêtes clientes est une véritable « ligne de front » entre le serveur et le monde extérieur. C'est pourquoi la sûreté de cette tâche est une considération critique pour la sécurité du serveur car tout comportement non désiré peut entraîner une faille de sécurité et l'accès illégal au serveur [95]. De plus, j'ai illustré précédemment que le décodage des requêtes clientes nécessite de prendre en compte le type d'E/S utilisé par le serveur (voir 3.3.3). En effet si les E/S sont non bloquantes, il est nécessaire de sauvegarder le contexte du décodage, puis de le restaurer à chaque fois, ce qui n'est pas le cas pour des E/S bloquantes.

La syntaxe d'un protocole de communication est typiquement spécifiée dans une RFC sous forme d'une succession de règles grammaticales. Ce type de spécification permet d'obtenir une machine à états finie qui est implantée de manière peu structurée. Le code ainsi obtenu est rarement exempt d'erreurs et difficilement maintenable.

Les décodeurs ou analyseurs de messages sont bien souvent construits à l'aide de générateurs d'analyseurs comme Yacc [53]. Cependant, ces outils ne sont pas conçus pour générer des analyseurs embarqués dans des serveurs. Il n'y a pas de support du décodage partiel des messages, pas de prise en compte des contraintes particulières au niveau de la gestion mémoire, i.e. aucune création à l'exécution, et ils ne prennent en compte que des E/S blo-

quantes. C'est pourquoi les analyseurs sont la plupart du temps implantés à la main. Pire, la complexité des analyseurs non bloquants est telle qu'ils sont systématiquement implantés de manière bloquante. Cette situation n'est plus acceptable du faite de la complexité, du nombre de nouveaux protocoles développés et de la perte potentielle des performances.

Afin de permettre de passer très facilement et de manière transparente d'un modèle à un autre, j'ai spécifié les principales caractéristiques nécessaire à un générateur d'analyseur syntaxique que je souhaitais incorporer à Saburo. Ces besoins (compatibles avec des E/S bloquantes et non bloquantes, gestion mémoire, etc.) ont guidé la réalisation d'un générateur d'analyseurs syntaxique nommé *Tatoo* [23] car aucun générateur d'analyseurs actuel [36, 60, 90] ne répondait à mes contraintes particulières. Les caractéristiques de Tatoo font qu'il est particulièrement désigné pour que les analyseurs syntaxiques qu'il génére soient embarqués dans des applications très performantes et à long cycle de vie comme un serveur Internet. A l'aide d'un langage spécifique au domaine propre à *Tatoo*, je décris la structure grammaticale des messages d'un protocole. Cette spécification est ensuite traitée par des générateurs qui permettent d'obtenir les tables d'analyse du langage que l'on souhaite analyser. Ces tables sont ensuite incorporées à une application à l'aide d'une bibliothèque dédiée.

Il existe d'autres générateurs d'analyseurs syntaxiques tels que DataScript [10], PacketType [77], PADS [35], GAPA [16], APG [67] et binpac [87] qui ont été récemment développés pour répondre au problème d'augmentation de la complexité des protocoles Internet. Cependant, ces outils ne prennent pas en compte tous les besoins des applications réseau en terme de performance et d'empreinte mémoire. Zebu [21] est un langage déclaratif, spécifique à un domaine, qui permet de décrire les messages d'un protocole de type HTTP. Zebu utilise comme entrée une grammaire EBNF étendue par des annotations. Ces annotations indiquent les champs de messages qui doivent être stockés dans des structures de données ainsi que d'autres informations sémantiques (le type de la valeur, les contraintes sur les valeurs possibles et si le champ est obligatoire ou optionnel). Une spécification Zebu est traitée par un compilateur afin de générer le code de traitement des messages réseaux qui sera ensuite utilisé dans une application. Une autre approche très similaire est le langage de spécification MSPL [33] (*My Simple Protocol Language*) implanté en Java. MSPL permet de décrire facilement un protocole Internet. Un compilateur va ensuite utiliser cette description pour générer les communications bas niveau et les parties liées au protocole aussi bien pour le client que pour le serveur. Cependant, Zebu et MSPL ne permettent pas de prendre en compte l'analyse non bloquante des requêtes et, à l'heure actuelle, ils ne sont pas intégrés dans un ensemble plus vaste permettant de produire intégralement des serveurs Internet fonctionnels.

Dans ce chapitre, je vais commencer par faire une brève introduction à la compilation et aux méthodes de génération automatique d'analyseurs. Dans une seconde partie, je vais décrire les principales caractéristiques que doivent posséder les analyseurs syntaxiques que l'on souhaite embarquer dans des serveurs Internet. Enfin, je vais présenter *Tatoo* un générateur d'analyseur conçu pour produire des analyseurs syntaxiques qui seront embarqués au sein de serveurs. En effet, *Tatoo* permet d'obtenir automatiquement des analyseurs bloquants et non bloquants, d'avoir une gestion mémoire dédiée aux applications à long cycle de vie et

de prendre les évolutions futures d'un langage. Je présenterais l'intégration de *Tatoo* au sein de Saburo.

5.1 Introduction à la compilation

Les principes et techniques utilisés lors du développement d'un compilateur sont réutilisables dans de très nombreux domaines. Ainsi, les grammaires non contextuelles sont utilisées pour implanter des systèmes de composition de textes et de dessins. Je vais utiliser ces techniques d'analyse de code source afin de décoder, dans un serveur Internet, les requêtes émises par un client.

5.1.1 Préliminaires

Tout d'abord, un compilateur (Fig. 5.1) peut être défini simplement :

Définition 5.1 *Un compilateur est un logiciel qui lit un programme écrit dans un langage source et le traduit en un programme équivalent écrit dans un langage cible [2].*

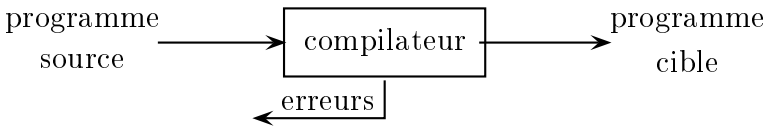


FIG. 5.1 – Vue simplifiée d'un compilateur

Conceptuellement, on peut distinguer deux grandes étapes dans le processus de compilation. L'*analyse* qui partitionne le programme en ses constituants et en crée une représentation intermédiaire. Et, la *synthèse* qui construit le programme cible désiré à partir de cette représentation intermédiaire. Cette seconde étape n'étant d'aucune utilité dans mon cas. Tout au long du processus de transformation, un rôle important du compilateur est de *signaler à son utilisateur la présence d'erreurs* dans le programme source.

5.1.1.1 L'analyse

La phase d'analyse d'un programme source va être constituée de trois étapes :

1. l'*analyse lexicale* qui constitue des *unités lexicales* à partir d'un flot de caractères ;
2. l'*analyse syntaxique* qui regroupe hiérarchiquement les unités lexicales afin de former des phrases ;
3. et enfin, l'*analyse sémantique* qui effectue certains contrôles pour assurer que l'assemblage des constituants a un sens.

En compilation, l'analyse sémantique permet de vérifier les types des différentes variables et leur compatibilité, qu'un opérande est bien compatible avec un opérateur, etc. Dans mon

cas particulier d'analyse de requête cliente dans un serveur Internet, les erreurs ne seront que lexicales et syntaxiques et la sémantique va correspondre à la « désérialisation » d'un objet représentant la requête cliente. La « désérialisation » correspond au remplissage des différents champs d'un objet. Puis, selon l'action demandée par le client, un service, codé *a posteriori* par le développeur, est fourni par le serveur Internet. Ce processus est donc très éloigné des techniques d'analyse sémantique classiques rencontrées en compilation.

5.1.2 L'analyse lexicale

La première étape du processus d'analyse d'un langage source, appelée *analyse lexicale*, est la lecture du flot d'entrée caractère par caractère afin de constituer une suite d'unités lexicales qui seront transmises et utilisées par l'analyseur syntaxique (Fig. 5.2).

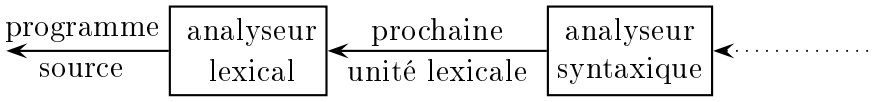


FIG. 5.2 – Interactions entre l'analyseur lexical et syntaxique

Remarque : L'analyseur lexical est vu comme un sous-programme ou une coroutine de l'analyseur syntaxique, i.e. il va lire, sur une demande explicite de l'analyseur syntaxique, les caractères en entrée pour identifier une unité lexicale.

Du fait d'une vision très localisée du programme source, il est difficile de détecter des erreurs au niveau de l'analyse lexicale. En effet, lors de l'analyse lexicale du programme :

```
fi( a == 1)
```

L'analyseur lexical ne peut pas déterminer s'il s'agit du mot clé `if` mal orthographié ou d'un identificateur de fonction non déclaré. Du fait de la validité de l'identificateur `fi`, l'analyseur lexical doit donc retourner l'unité lexicale d'un identificateur et laisser le soin de traiter cette erreur éventuelle à une autre phase du compilateur, l'analyseur syntaxique.

Remarque : Il peut arriver que l'analyseur lexical ne puisse plus continuer son analyse, i.e. aucune règle d'unité lexicale ne correspond au préfixe du flot d'entrée. Il est alors nécessaire de mettre en place des *mécanismes de récupération sur erreur* au niveau de la phase d'analyse lexicale [2].

5.1.2.1 Pourquoi séparer la phase d'analyse ?

Séparer la phase d'analyse en deux parties, analyse lexicale et analyse syntaxique, permet une *conception plus simple et plus propre* d'un compilateur. Cela permet aussi d'améliorer l'*efficacité* d'un compilateur. En effet, une grande partie du temps de compilation est dédiée à la lecture du flot d'entrée et à son découpage en unités lexicales. Il existe différentes techniques, dont la gestion de tampons de données, qui peuvent être exploitées spécifiquement dans l'analyseur lexical pour optimiser cette phase de lecture. Enfin, cela permet d'améliorer la *portabilité* du compilateur. En effet, le codage des caractères du flot d'entrée ou la représentation de codages spéciaux d'un langage ne sont traités qu'au niveau de l'analyseur lexical sans aucune implication dans les autres parties du compilateur. Il suffira de changer l'analyseur lexical pour prendre en compte le changement du codage des caractères.

5.1.2.2 Construction d'un analyseur lexical

Bien que conceptuellement moins complexe que les autres phases de la compilation, le temps d'exécution de l'analyse lexicale est un facteur très important lors de la conception d'un compilateur.

L'analyse lexicale est la seule phase d'un compilateur à lire le programme source caractère par caractère !

Actuellement, il existe trois grandes approches de développement d'un analyseur lexical. La première est de développer *directement en langage d'assemblage* et de gérer explicitement la lecture du flot d'entrée. Une seconde approche est d'*utiliser un langage de programmation système* et d'exploiter les services d'E/S de ce langage afin de lire le flot d'entrée. Enfin, on peut *utiliser un générateur d'analyseur lexical* pour le concevoir automatiquement. Généralement, le générateur d'analyseur lexical va utiliser un ensemble d'expressions régulières qui vont décrire les règles lexicales de l'analyseur que l'on souhaite produire. De plus, c'est au générateur de fournir les routines de lecture et de mémorisation du flot d'entrée.

Il existe de nombreux générateurs d'analyseurs lexicaux et syntaxiques produisant des analyseurs dans différents langages de programmation et avec des fonctionnalités plus ou moins complexes. Dans la section 5.2, je vais présenter *Tatoo* [23] l'un de ces outils et, *via* l'exemple d'un serveur HTTP simplifié (section 5.3.1), je vais montrer comment les construires dans le cas d'un serveur Internet.

5.1.3 L'analyse syntaxique

La seconde étape du processus d'analyse, appelée *analyse syntaxique*, interagit avec l'analyseur lexical afin d'obtenir une chaîne d'unités lexicales et vérifie que cette chaîne soit bien engendrée par la grammaire du langage source (Fig. 5.3).

Une caractéristique très importante d'un analyseur syntaxique est sa capacité à *signaler chaque erreur de syntaxe de manière précise et intelligible* afin de permettre la correction du programme source. En effet, beaucoup d'erreurs sont par nature syntaxiques ou sont

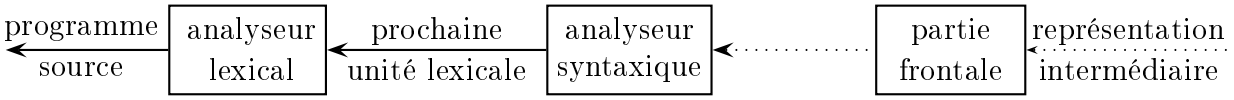


FIG. 5.3 – Interactions entre analyseur syntaxique et son environnement

révélées lorsque le flot d'unités lexicales provenant de l'analyseur lexical contredit les règles grammaticales définissant le langage de programmation. De plus, la précision des méthodes d'analyse actuelles permet de détecter très efficacement la présence d'erreurs syntaxiques dans les programmes.

Remarque : La détection précise de la présence d'erreurs sémantiques ou logiques au moment de la compilation est une tâche bien plus difficile [2].

5.1.3.1 Les grammaires

Dans le but de décrire la structure syntaxique d'un programme ou d'un protocole de communication, on utilise très souvent des grammaires non contextuelles ou notation BNF (Backus-Naur Form). L'utilisation des grammaires offre de nombreux avantages lors du développement d'un compilateur. En effet, elles offrent une *syntaxe précise et facile à comprendre* d'un langage ou d'un protocole. De plus pour certaines classes de grammaires, il est possible de *construire automatiquement des analyseurs syntaxiques* efficaces et, lors de cette construction, il est possible de *détecter automatiquement des ambiguïtés* syntaxiques et de les corriger très rapidemment. Enfin, il est plus facile de *suivre les évolutions d'un langage* par cette approche car il suffit d'ajouter, à la grammaire déjà existante, les constructions introduites par la nouvelle version du langage ou du protocole.

Les constructions des langages de programmation présentent une structure récursive qui peut être définie par une *grammaire non contextuelle*. Celle-ci est formée de terminaux, de non-terminaux, d'un axiome et de productions.

1. les *terminaux* sont les symboles de base à partir desquels les chaînes sont formées ;
2. les *non-terminaux* sont des variables syntaxiques qui définissent des ensembles de chaînes aidant à spécifier le langage engendré par la grammaire ;
3. l'*axiome* est un non-terminal particulier tel que l'ensemble des chaînes. Il dénote correspond exactement au langage défini par la grammaire ;
4. les *productions* spécifient la manière dont les terminaux et les non-terminaux peuvent être combinés pour former des chaînes.

Ainsi, chaque production va être constituée d'un non-terminal dit partie gauche de la production, suivie d'une flèche, suivie d'une chaîne formée de terminaux et de non-terminaux dite partie droite.

5.1.3.2 Méthode d'analyse syntaxique

Actuellement, trois méthodes générales d'analyse syntaxique sont discernables. Les *méthodes universelles* Cocke-Younger-Kasami [26, 55, 115] et la méthode d'Earley [34] qui sont capable d'analyser n'importe quelle grammaire mais elles sont peu efficaces (complexité en $O(n^3)$). Les méthodes *descendantes*, i.e. les arbres d'analyse sont construits du haut (la racine) vers le bas (les feuilles). Et enfin, les méthodes *ascendantes* pour lesquelles les arbres d'analyse sont construits des feuilles vers la racine. Ces deux dernières méthodes sont les plus couramment utilisées en compilation (complexité en $O(n)$) mais ne fonctionnent que pour certaines classes de grammaires [2].

Il faut noter que les méthodes descendantes et ascendantes sont toutes les deux parfaitement réutilisables. Cependant, l'implantation de l'approche descendante est basée en général sur des appels récursifs ce qui pose problème dans le cas d'appel d'E/S bloquant.

5.1.3.3 Table d'analyse syntaxique

Un programme d'analyse repose sur un diagramme de transitions et va essayer de faire correspondre des transitions du diagramme avec les symboles de l'entrée et simuler un appel de procédures à chaque fois qu'il doit suivre un arc étiqueté par un non-terminal.

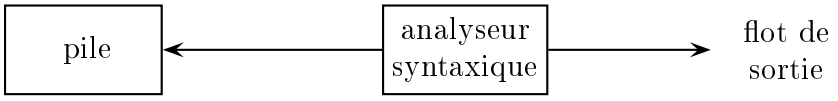


FIG. 5.4 – Fonctionnement d'un analyseur syntaxique

En résumé, pour un langage source donné et une méthode d'analyse (ascendante ou descendante), la principale difficulté dans la construction d'un analyseur syntaxique va résider dans l'obtention de tables d'analyse, seule différence d'un analyseur à un autre.

5.2 Générateur d'analyseurs pour serveurs Internet

Précédemment, j'ai montré que les analyseurs pouvaient être générés automatiquement à l'aide d'outils dédiés (section 5.1.2.2). En effet, la principale différence entre deux analyseurs est leur table d'analyse. Les générateurs d'analyseurs vont produire de manière automatique ces tables à partir d'une description sous forme de grammaire d'un langage ou d'un protocole de communication.

Pour embarquer des analyseurs syntaxiques au sein de mon modèle de développement de serveurs Internet, j'ai dû définir un certain nombre de contraintes. En tout premier lieu, la principale motivation d'utiliser un générateur d'analyseur syntaxique dans Saburo est de fournir des analyseurs capables de travailler en présence d'E/S non bloquantes (section 5.2.1). Une seconde caractéristique, extrêmement intéressante est la prise en compte de l'encodage

des caractères (section 5.2.2) avec un minimum de recopies. Enfin, il est nécessaire de fournir une gestion de tampons de données très performante ainsi qu'un faible taux de création d'objets à l'exécution (section 5.2.3). En effet, la grande majorité des instanciations sont faites à l'initialisation du système et non pas à l'exécution.

5.2.1 Analyse lexicale et syntaxique non bloquante

Les principaux générateurs d'analyseurs syntaxiques actuels [36, 60, 90] ne fournissent que des analyseurs lexicaux et syntaxiques compatibles avec des E/S bloquantes. Ce type d'analyseurs va *extraire* les données depuis un flot d'entrées et *attendre* la disponibilité de nouvelles données sur ce flot (Fig. 5.5).

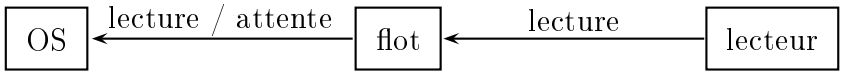


FIG. 5.5 – Analyse de texte par « traction » (*pull*)

Ce comportement d'extraction de données va entraîner des *périodes d'attente* qui sont *actives* et *bloquantes*. Lors de la lecture du mot w d'un langage, l'analyseur lexical peut être arrêté à un préfixe p_w de ce mot car la suite du mot w n'est pas encore disponible sur le flot d'entrée. Pendant une période d'attente t_w , l'analyseur lexical va attendre activement qu'un facteur f_w du mot w soit disponible en entrée (Fig. 5.6)

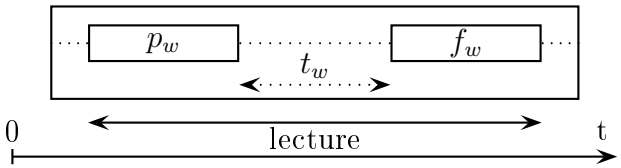


FIG. 5.6 – Attente active des analyseurs

Dans le cas d'applications performantes, ce comportement est acceptable si ces périodes d'attente sont de faibles durées (analyse de fichiers présents sur un disque dur). Si les données proviennent d'une connexion réseau, ce comportement n'est plus satisfaisant car les périodes d'attente sont alors beaucoup trop importantes. C'est pourquoi et afin de fournir une analyse lexicale et syntaxique efficace dans le cas de connexions réseau, il est nécessaire de produire du code *réveillant* et *transmettant* les données, quand celles-ci sont disponibles, aux analyseurs lexicaux et syntaxiques.

Dans ce type de comportement, le processus d'analyse est démarré explicitement quand de nouvelles données sont disponibles et arrêté lorsqu'il n'y a plus de données sur le flot d'entrée (Fig. 5.7).

Comme les processus d'analyse lexicale et syntaxique vont être régulièrement démarrés et arrêtés selon les disponibilités des données sur le flot d'entrée, il est nécessaire de conserver un état afin de sauvegarder les traitements en cours. Lors de la lecture d'un mot w du langage, l'analyseur lexical peut être arrêté à un préfixe p_w de ce mot car la suite du mot w n'est pas

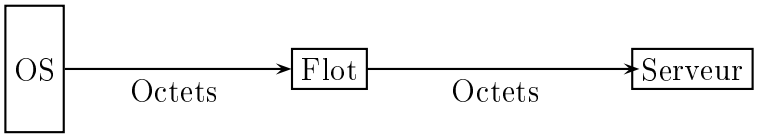


FIG. 5.7 – Analyse de texte par « poussée » (*push*)

encore disponible en entrée. Après une période d'attente t_w, l'analyseur lexical sera réveillé lorsque un facteur f_w du mot w sera nouvellement disponible sur le flot d'entrée (Fig. 5.8).

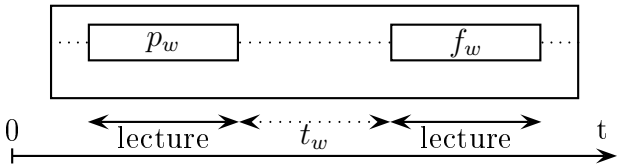


FIG. 5.8 – Attente passive des analyseurs

Pratiquement, l'implantation des analyseurs lexicaux et syntaxiques doit être basée principalement sur deux méthodes :

1. Une méthode (par exemple, step()) qui est utilisée pour réveiller et transmettre les caractères et terminaux aux analyseurs.
2. Et une méthode (par exemple, close()) qui va indiquer la fin du processus d'analyse.

Cependant, il est important de noter que la fin du processus d'analyse ne correspond pas obligatoirement à la fermeture de la connexion client-serveur. En effet, dans le cas du protocole HTTP, les requêtes peuvent être chaînées par le client. Pour déterminer si la fin du processus d'analyse est atteinte, il est nécessaire de spécifier une fin valide pour les requêtes analysées. Une fois cette fin atteinte, le processus d'analyse doit être réinitialisé pour traiter la requête suivante.

Remarque : Il est possible de simuler le comportement d'extraction de données à l'aide du comportement de transmission de données en utilisant la boucle read/step suivante :

```
while( !endOfStream())
    step() ;
close() ;
```

La condition d'arrêt est la fin du flot d'entrée. L'analyseur lexical sera ensuite fermé et aura la charge de fermer l'analyseur syntaxique.

5.2.2 Encodage des caractères

Dans le but de fournir le support de plusieurs jeux de caractères, un analyseur syntaxique peut utiliser le mécanisme de flot de caractères fourni par le langage (par exemple, java.io.Reader en Java). Cependant, il est aussi possible d'encoder les tables lexicales directement dans le jeu de caractères du flot d'entrée. Ce mécanisme a l'avantage d'éviter les opérations de codage et décodage des caractères à l'exécution et les recopies. Il est donc plus efficace mais le jeu de caractères doit être restreint et l'implantation de l'analyseur lexical est fortement liée à un jeu de caractères particulier. De plus, la spécification des intervalles dans les expressions régulières va elle aussi dépendre de l'encodage (ASCII, Unicode, ...) des caractères. Cependant, il est possible d'utiliser ces intervalles lorsque les lettres et les chiffres ne dépendent pas du jeu de caractères.

Une autre optimisation dépendante du jeu de caractères est l'implantation de l'automate des règles de l'analyseur lexical. Typiquement, l'implantation des transitions de cet automate utilise la notion d'intervalles de caractères. Dans le cas de l'encodage Unicode des caractères ces intervalles ne peuvent pas être directement implantés sous forme de tableaux. Des implantations plus efficaces de cet automate des transitions peuvent être fournies :

1. par des tableaux, lorsque l'ensemble des caractères n'est pas important ;
2. par une suite d'instructions switch-case.

La seconde implantation est intéressante quand la plupart des transitions peuvent être implantées par le cas default.

5.2.3 Gestion de la mémoire

L'une des principales caractéristiques d'un analyseur syntaxique embarqué est de prévenir au maximum les créations d'objets durant l'exécution. C'est pourquoi, la quasi totalité des objets doit être allouée à l'initialisation et aucune autre allocation ne doit être réalisée durant l'exécution, sauf quelques extensions potentielles de la pile d'analyse syntaxique. Ce comportement est extrêmement important pour permettre d'incorporer les analyseurs générés dans des applications performantes et à très longue durée de vie, comme un serveur Internet. De même, les analyseurs doivent être réutilisables *via* des viviers d'objets et les tables partagées pour limiter la consommation mémoire et le surcoût induit par la création de nouveaux objets.

5.3 Tatoo, un générateur d'analyseurs dédié

A partir des caractéristiques nécessaires pour l'embarquement d'un analyseur au sein d'un serveur Internet, j'ai étudié les différents générateurs d'analyseurs syntaxiques en Java existants [31, 36, 39, 49, 60, 90] mais aucun ne répondait pas à mes besoins. C'est pourquoi, j'ai participé à l'élaboration des spécifications de *Tatoo* [23] afin de pouvoir embarquer dans un serveur Internet les analyseurs syntaxiques qu'il génére.

La motivation initiale de *Tatoo* est de fournir un générateur d'analyseur syntaxique compatible avec des E/S non bloquantes [23]. Les analyseurs syntaxiques générés peuvent être des analyseurs ascendants (LALR, LR ou SLR) ou descendants (LL).

Je vais maintenant présenter *Tatoo* [23] qui, à partir d'un ensemble d'expressions régulières décrivant les unités lexicales, d'une spécification formelle de la grammaire du langage et de plusieurs règles sémantiques, est capable de générer automatiquement un analyseur syntaxique. Puis je vais présenter l'intégration de *Tatoo* dans Saburo. Je finirais par une liste non exhaustive de générateurs d'analyseur syntaxique existant actuellement.

5.3.1 Construction d'un analyseur à l'aide de *Tatoo*

Je vais maintenant présenter la construction d'un analyseur à l'aide de *Tatoo* pour un protocole HTTP simplifié. Ce processus est découpé en plusieurs phases :

1. spécification des règles d'analyse du langage (ici le protocole HTTP) ;
2. génération automatique des différents analyseurs (lexical et syntaxique) ;
3. implantation manuelle de la sémantique.

Il est important de noter que cette dernière étape est dépendante du contexte dans lequel nous utilisons les analyseurs générés par *Tatoo*. Dans le contexte de l'analyse des requêtes clientes d'un serveur Internet, cette phase va consister à la « désérialisation » de la requête en un objet Java.

5.3.1.1 Spécification des règles d'analyse d'un langage

La spécification d'un langage est faite sous forme d'un fichier EBNF (*Extended Backus-Naur Form*). Ce fichier est séparé en une partie lexicale (unités lexicales), une partie syntaxique (productions) et une partie sémantique (lien avec les objets Java sous-jacents).

Grammaire du protocole HTTP

Selon la norme OSI [51], l'HyperText Transfer Protocol [52] est un protocole de niveau application. Il est utilisé pour la transmission et la récupération de documents distribués et multimédias. Les fonctionnalités du protocole HTTP sont, outre la récupération de données, la possibilité d'effectuer des recherches, des fonctions de remise à jour et d'annotations de documents. Le protocole de communication HTTP est complètement décrit par une grammaire non contextuelle mais, par souci de simplification, je vais me restreindre à la description de la grammaire pour la seule méthode `GET`.

La requête émise par un client vers un serveur inclut dans sa première ligne :

1. la méthode appliquée à la ressource ;
2. l'identificateur de cette ressource ;
3. la version du protocole utilisée par le client.

Dans le but d'assurer une compatibilité descendante avec la version HTTP/0.9 du protocole, deux formats de requête sont valides.

$$
\begin{array}{lll}
\textit{Request} & \rightarrow & \textit{SimpleRequest} \mid \textit{CompleteRequest} \\
\textit{SimpleRequest} & \rightarrow & \text{“}\textit{GET}\text{”}\ \textit{URI}\ \textit{CRLF} \\
\textit{CompleteRequest} & \rightarrow & \textit{RequestLine} \\
 & & (\textit{Head} \mid \textit{RequestHead} \mid \textit{EntityHead}) * \textit{CRLF} \\
 & & [\textit{EntityBody}\,]
\end{array}
$$

Lorsqu'un serveur HTTP reçoit une requête simple, il devra obligatoirement répondre par une réponse simple. De plus, le protocole stipule qu'un client capable de recevoir une réponse complète ne devra jamais émettre de requête simple.

La ligne de requête commence par le nom d'une requête, suivie de l'URI de la ressource, du numéro de version du protocole utilisé, et se termine par la suite de caractère CRLF. Ces éléments sont séparés par des espaces et aucun CR ni LF n'est autorisé excepté la séquence finale CRLF.

$$\textit{RequestLine} \rightarrow \textit{Method}\ \textit{URI}\ \textit{Version}\ \textit{CRLF}$$

La méthode indiquée en tête de ligne est destinée à être appliquée à l'URI cible. Son nom est dépendant de la casse des caractères. Dans notre cas, nous nous restreignons à la seule méthode GET, le protocole HTTP définit d'autres méthodes telle que les méthodes HEAD, POST, PUT...

$$\textit{Method} \rightarrow \text{“}\textit{GET}\text{”}$$

L'URI visée est l'URI identifiant la ressource réseau à laquelle doit être appliquée la méthode.

$$\textit{URI} \rightarrow \textit{AbsoluteURI} \mid \textit{AbsolutePath}$$

Le chemin absolu ne doit pas être vide. Si la ressource se trouve dans la racine, le chemin spécifié devra comporter au moins le caractère slash ('/').

Les messages de requête et de réponse contiennent généralement une entité dans laquelle est incluse des éléments de requête ou de réponse. Une entité va être définie par son en-tête, et dans la plupart des cas par un corps.

$$
\begin{array}{lll}
\textit{EntityHead} & \rightarrow & \textit{Content-Encoding} \\
 & \mid & \textit{Content-Length} \\
 & \mid & \textit{Content-Type} \\
 & \mid & \ldots
\end{array}
$$

Le champ Content-Encoding est utilisé pour compléter l'information du type de média. Lorsqu'il est présent, il indique sous quel codage la ressource est enregistrée.

Content-Encoding → *"Content-Encoding : " CodingType CRLF*

Le champ d'en-tête Content-Length indique la taille du corps d'entité, sous la forme d'un nombre d'octets exprimé en décimal.

Content-Length → *"Content-Length : " 1*DIGIT*

Le champ d'en-tête Content-Type indique le type de média envoyé au récepteur dans le corps d'entité.

Content-Type → *"Content-Type : " MediaType*

Le corps d'entité (s'il existe) envoyé dans un message de requête ou de réponse HTTP est dans un format et sous un encodage défini par les champs d'en-tête d'entité.

EntityBody → *OCTET**

Remarque : Un corps d'entité n'apparaît dans un message de requête que dans la mesure ou le type de la requête le demande. La présence de ce corps est signalée par la présence d'un champ Content-Length dans les champs d'en-tête de la requête.

Grammaire EBNF Tatoo du protocole HTTP

La grammaire EBNF Tatoo va être composée de plusieurs parties. La première, tokens, donne l'ensemble des lexèmes reconnu par l'analyseur lexical. La seconde, blancks, spécifie les caractères qui sont automatiquement consommée par le processus d'analyse (par exemple une succession d'espaces). La troisième, starts définie la ou les production(s) initiale(s). Enfin, la dernière partie, productions, indique l'ensemble des productions de la grammaire.

```
tokens :
    service='GET|POST'
    url= '([/])+'
    httpslash= 'HTTP\/'
    version= '([0-9])'
    colon= ' :'
    h_key= '([^ :\r\n])+'
    keepalive= 'Keep-Alive'
    h_value= '[^\r\n]([^\r\n])+'
    eoln= '(\r)?\n'
    dot= '.'

blanks :
    space= "( |\t)+"

starts :
    start

productions :
    start = request+ { start } ;
```

```
request = firstline 'eoln' header* 'eoln' { request } ;

firstline = 'service' 'url' 'httpslash' 'version' 'dot' 'version' { firstline } ;

header = 'h_key' 'colon' headervalue 'eoln' { header } ;

headervalue = 'h_value' { headervalue } | 'keepalive' { keepalive } ;
```

5.3.1.2 Génération automatique des analyseurs

Tatoo fournit des générateurs pour produire le code de n'importe quel type de langage. En particulier, il fournit nativement des générateurs pour le langage Java. Les générateurs de code Java sont basés sur velocity [8] un paquetage d'Apache qui utilise une approche modèle-vue pour la génération de fichiers. Les mécanismes de *Tatoo* produisent toutes les tables nécessaires à l'écriture d'un analyseur lexical et syntaxique et les interfaces nécessaires à l'implantation de la sémantique.

La spécification de la grammaire et des terminaux sous forme d'EBNF va être utilisée pour :

- la génération automatique d'un « écouteur » d'analyse lexicale,
- l'ensemble des tables d'analyse lexicale et syntaxique,
- les interfaces GrammarEvaluator (voir annexe B.1) et TerminalEvaluator (voir annexe B.2)

Ces deux interfaces sont utilisées pour l'implantation de la sémantique. Le développeur va spécifier manuellement une sémantique particulière pour les attributs terminaux en donnant une implantation de cette interface. Il est ainsi possible de spécifier différentes sémantiques sans regénérer à chaque fois les différentes classes d'analyse. De plus, des classes utilitaires (Analyzer et AnalyzerBuilder) sont générées afin de simplifier la construction, l'utilisation et le paramétrage des analyseurs lexicaux et syntaxiques.

5.3.1.3 Implantation de la sémantique

L'implantation de la sémantique dans le cas du serveur HTTP simplifié correspond à l'implantation des deux interfaces générées par *Tatoo*. Cette implantation va correspondre à la sérialisation du flot des lexèmes récupérés lors de la phase d'analyse en un objet Java. Cet objet, nommé Request, va représenter la requête du client au serveur. L'implantation de l'interface TerminalEvaluator va être la suivante :

```
public class HttpTerminalEvaluator implements TerminalEvaluator< ByteBuffer > {
  /**
  * This method is called when the rule <code>get</code> is recognized
  *
  * @return the value associated with the terminal spawn for the rule
  */
  public final Method get(ByteBuffer data) {
    return Method.GET ;
  }
```

```
  /**
  * This method is called when the rule <code>header_key</code> is recognized
  *
  * @return the value associated with the terminal spawn for the rule
  */
  public final String header_key(ByteBuffer data) {
    return extractString(data) ;
  }

  /**
  * This method is called when the rule <code>header_value</code> is recognized
  *
  * @return the value associated with the terminal spawn for the rule
  */
  public final String header_value(ByteBuffer data) {
    return extractString(data) ;
  }

  /**
  * This method is called when the rule <code>url</code> is recognized
  *
  * @return the value associated with the terminal spawn for the rule
  */
  public final String url(ByteBuffer data) {
    return extractString(data) ;
  }

  /**
  * This method is called when the rule <code>http09</code> is recognized
  *
  * @return the value associated with the terminal spawn for the rule
  */
  public final Version http09(ByteBuffer data) {
    return Version.HTTP_0_9;
  }

  /**
  * This method is called when the rule <code>http10</code> is recognized
  *
  * @return the value associated with the terminal spawn for the rule
  */
  public final Version http10(ByteBuffer data) {
    return Version.HTTP_1_0;
  }

  /**
  * This method is called when the rule <code>http11</code> is recognized
  *
  * @return the value associated with the terminal spawn for the rule
  */
  public final Version http11(ByteBuffer data) {
    return Version.HTTP_1_1;
  }
}
```

Cette implantation permet de spécifier le comportement de l'analyseur pour chaque terminal rencontré. Lorsque l'analyseur rencontrera le lexème GET, il passera par cette implantation de l'interface TerminalEvaluator pour connaître l'opération à effectuer, ici le retour de la valeur GET de l'énumération Method. Pour d'autres terminaux, il sera nécessaire de convertir le flot d'octets en chaîne de caractères (la méthode extractString). Tatoo permet ainsi de minimiser les opérations de conversion et de recopie. Maintenant, il est nécessaire de spécifier le comportement sémantique pour les productions reconnues par l'analyseur. C'est à dire l'implantation de l'interface GrammarEvaluator :

```
public class HttpGrammarEvaluator implements GrammarEvaluator {
  private Request req;

  public final void restart(Request req) {
    this.res = res;
  }

  /**
  * This methods is called after the reduction of the non terminal method
  * by the grammar production method_get.
  */
  public final Method method_get(Method get) {
    return get;
  }

  /**
  * This methods is called after the reduction of the non terminal firstline
  * by the grammar production firstline.
  */
  public final void firstline(Method method, String url, Version version) {
    switch(method) {
    case GET :
      req.setMethodRequest(Method.GET);
      req.setUrlRequest(url);
      req.setVersionRequest(version);
      break;
    }
  }

  /**
  * This methods is called after the reduction of the non terminal version
  * by the grammar production version_http09.
  */
  public final Version version_http09(Version http09) {
    return http09;
  }

  /**
  * This methods is called after the reduction of the non terminal version
  * by the grammar production version_http10.
  */
  public final Version version_http10(Version http10) {
    return http10;
  }

  /**
  * This methods is called after the reduction of the non terminal version
  * by the grammar production version_http11.
  */
```

```
  public final Version version_http11(Version http11) {
    return http11 ;
  }

  public void header_part(String header_key, String header_value) {
    req.setHeader(header_key, header_value) ;
  }

  public void endline() {
    // do nothing
  }

  public void request() {
    // do nothing
  }

}
```

L'objet Request correspond à la requête du client sérialisée par le processus d'analyse. Cette objet est similaire aux *beans* Java [105] car il implante seulement des méthodes pour fixer et récupérer des informations. Dans cette implantation, seulement deux méthodes vont réellement être opérante :

- quand la réduction de la production firstline se produit ;
- pour la réduction de la production header_part.

L'opération réalisée par ces deux méthodes est la sauvegarde d'informations dans l'objet Request. Mais, comment utiliser toutes ces spécifications ?

La solution la plus simple pour utiliser l'analyseur consiste à appeler une méthode statique run() de la classe Analyzer générée par Tatoo. Cependant cette méthode n'est pas assez précise car il est impossible de mettre en place un gestionnaire d'erreurs au niveau lexical ou syntaxique car elle utilise des gestionnaires d'erreurs fournis par défaut. Dans le cas d'un serveur HTTP si la requête présente des mots mal orthographiés, i.e. qui ne correspondent pas aux lexèmes souhaités, ou des phrases avec des erreurs grammaticales, le serveur doit rejeter la requête et fermer la connexion. Les gestionnaires fournis par défaut tentent de faire une reprise sur erreur en recherchant par exemple, le premier mot correct. Ce qui ne correspond pas au comportement souhaité !

5.3.2 Intégration de Tatoo dans un serveur HTTP simplifié

La première étape du développement du serveur HTTP est de spécifier le graphe des stages. Je vais réutiliser le même graphe de stage que celui donné dans la section 3.3. Je rappelle que les connexions entre les stages représentent les synchronisations et les communications dans le serveur HTTP. Selon son type, un stage peut envoyer (vers son ou ses successeurs) et recevoir (de son ou ses prédecesseurs) des événements de communication.

Le serveur HTTP utilisé comme exemple est composé de six stages (voir section 5.3.2).

Seul le stage de décodage de la requête cliente va être modifié par l'utilisation de *Tatoo*, modifications que je vais maintenant présenter.

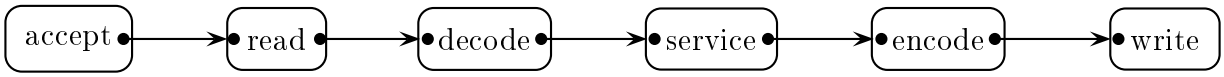


FIG. 5.9 – Graphe de connexion des stages du serveur HTTP

5.3.2.1 Spécification des événements de communication

Afin d'échanger des informations entre les stages et selon leur position dans le graphe, il est nécessaire de spécifier des événements d'entrée et de sortie. Ces événements sont utilisés pour envoyer et recevoir des informations entre les différents stages. L'intégration de *Tatoo* dans le développement du serveur HTTP entraîne la modification de l'événement d'entrée du stage de décodage.

L'événement d'entrée du stage de décodage devient :

```
public interface DecodeStageRequest {
  public HttpSaburoLexer getHttpSaburoLexerRequest();
  public ByteBuffer getByteBufferRequest();
}
```

La classe HttpSaburoLexer est une classe d'encapsulation qui permet de simplifier l'utilisation des analyseurs fournis par *Tatoo*. Cette classe fournit trois méthodes utilitaires :

1. hasRemaining() qui détermine si le tampon de données sous-jacent contient encore des octets.
2. step() : qui traite tous les caractères présents dans le tampon de données ;
3. endOfParsing() qui détermine si on est arrivé à la fin du flot ou d'une requête.

La méthode endOfParsing() est très particulière car elle dépend en grande partie du protocole utilisé. Ainsi dans le cas du protocole HTTP, il est possible d'avoir des connexions persistantes, c'est-à-dire que les requêtes sont considérées comme appartenant à un flot de requêtes. Dans ce cas, le serveur doit traiter une requête, puis passer à la suivante sans fermer la connexion. Cependant, comment peut-on savoir si on a fini d'analyser une requête pour passer à la suivante ? La méthode endOfParsing() est le point d'entrée qui permet de réaliser ce type d'opération. Dans le cas du protocole HTTP et de la grammaire donnée précédemment, la méthode endOfParsing va récupèrer l'ensemble des terminaux possibles que l'analyseur syntaxique peut reconnaître. Puis, déterminer si le seul terminal possible est le double CRLF marquant la fin d'une requête HTTP. Si c'est le cas alors on a fini d'analyser la requête. Cette méthode sera codée de la façon suivante :

```
public boolean endOfParsing() {
  Set<? extends TerminalEnum > lookahead = parser.getLookahead();
```

```
    if((lookahead.size() == 1) && (lookahead.contains(TerminalEnum.__eof__)))
     return true ;
   return false ;
}
```

5.3.2.2 Spécification du stage de décodage

Je vais maintenant implanter le stage de décodage. Je rappelle qu'un stage est une suite d'instructions avec au plus un appel d'E/S et sans aucun bloc de synchronisation.

Le stage de décodage est implanté de la façon suivante :

```
@Stage
public class DecodeStage< Q extends DecodeStageRequest, S extends DecodeStageResponse > {
  public final void handle(StageContext< S > ctx, Q req, S res) {
    HttpSaburoLexer lexer = req.getHttpSaburoLexerRequest() ;
    while(lexer.hasRemaining()) {
      lexer.step() ;
    }
    lexer.compact() ;
    if(lexer.endOfParsing()) {
      ctx.dispatchToSuccessor(res) ;
      lexer.reset() ;
    }
  }
}
```

Le stage de décodage va analyser la requête au fur et à mesure de l'arrivée des caractères et ce tant qu'il reste des caractères dans le tampon de lecture. Lorsque l'analyse est terminée la méthode compact permet de copier les caractères qui n'ont pas été reconnus en début de tampon, ce qui facilitera sa réutilisation. On teste si on est arrivé à la fin d'une requête et si tel est le cas, l'objet DecodeStageResponse qui regroupe toutes les informations sérialisées à l'aide de *Tatoo via* l'interface GrammarEvaluator est transmise au(x) successeur(s) du stage de décodage. Enfin, l'analyseur est réinitialisé pour pouvoir reconnaître une nouvelle requête.

5.3.3 Travaux similaires

Actuellement, il existe un grand nombre de générateurs d'analyseurs écrits dans différents langages de programmation. Présenter exhaustivement l'ensemble de ces outils sortirait complètement du cadre de ce travail. C'est pourquoi, je vais me restreindre à la présentation des générateurs d'analyseurs écrits en Java et qui produisent des analyseurs en Java.

Java Compiler Compiler [60] est un générateur d'analyseurs syntaxiques descendants écrit en Java et développé par Sun Microsystems. JavaCC fournit des analyseurs syntaxiques de type LL(k), c'est-à-dire qu'il est nécessaire d'utiliser une fenêtre d'analyse de taille k pour résoudre les ambiguïtés. Contrairement à *Tatoo*, la partie sémantique de l'analyseur est directement intégrée dans la spécification de la grammaire. Ce couplage fort entre grammaire et sémantique implique un risque important d'erreurs dans le cas de grande grammaire (celle d'un langage de programmation) et ne permet pas de réutiliser une grammaire indépendamment de sa sémantique.

ANother Tool for Language Recognition [90] est aussi un générateur d'analyseurs syntaxiques descendants. Similairement à *JavaCC*, *ANTLR* fournit des analyseurs syntaxiques de type LL(k) et la grammaire du langage est étendue par des actions de sémantique. Pour la construction, le parcours et la transformation de l'arbre de syntaxe abstrait, *ANTLR* fournit un langage spécifique au domaine.

JFlex [39] et *Cup* [49] sont des implantations en Java de *Lex* [65] et *Yacc* [53]. *JFlex* est un générateur en Java qui réutilise le même format d'expressions régulières que *Lex*. *Cup* génère des analyseurs ascendants à partir de grammaires LALR(1) en utilisant une syntaxe similaire à *Yacc*. De même que *JavaCC* et *ANTLR*, *Cup* étend la grammaire en incluant des actions de sémantique spécifiées en Java.

Sable Compiler Compiler [36] est un générateur d'analyseurs descendants à partir de grammaires LALR(1). Cependant, les conflits ne sont pas résolus par l'associativité ou la spécification de propriétés. Contrairement aux trois générateurs précédents, *SableCC* ne permet pas de spécifier de sémantique dans les grammaires mais, va générer l'arbre de syntaxe abstrait et utiliser le patron de conception visiteur [37] afin de spécifier une sémantique donnée. Il y a donc une séparation claire entre la spécification de la grammaire et la partie sémantique. Cependant, il n'est pas possible de générer un arbre spécifique, par exemple un arbre différent selon la version du langage, sans devoir recréer un nouvel arbre de syntaxe abstrait.

Enfin, *Beaver* [31] est un générateur d'analyseurs prenant en entrée des grammaires LALR(1). La motivation première de *Beaver* est de générer des analyseurs très rapides. De même que *Tatoo*, *Beaver* va générer une table d'analyse syntaxique qui contient les actions à réaliser. Les appels de fonctions sont basés sur le choix de la fonction à l'exécution en fonction du type réel de l'action. C'est un mécanisme de *late-binding* qui surpasse en temps d'exécution les implantations traditionnelles de type `switch-case`.

Remarque : Aucun des générateurs présentés ci-dessus ne génèrent des analyseurs capablent de fonctionner en présence d'E/S non bloquantes.

5.4 En conclusion...

Dans un serveur Internet, le décodage des requêtes clientes est l'interface d'interaction entre le serveur et ses clients. La sûreté et la robustesse de ses traitements sont primordiaux lors de la conception d'un serveur. En effet, tout comportement non désiré peut permettre des accès illégaux au serveur ou empêcher de servir correctement un client. De plus, le décodage des requêtes clientes nécessitent de prendre en compte le type d'E/S utilisées par le serveur. En effet si les E/S sont non bloquantes, il est nécessaire de sauvegarder le contexte du décodage, puis de le restaurer à chaque fois, ce qui n'est pas le cas pour des E/S bloquantes.

Actuellement, la syntaxe d'un protocole de communication est typiquement spécifiée dans une RFC sous forme d'une succession de règles grammaticales (EBNF). Ce type de spécification permet d'obtenir une machine à états finie qui peut être implantée de manière peu structurée. Le code obtenu manuellement n'est ni exempt d'erreurs, ni facilement maintenable. Cependant à partir d'une grammaire, il est possible de construire automatiquement des analyseurs lexicaux et syntaxiques mais, ces analyseurs sont bien souvent construits à l'aide de générateurs qui n'ont pas été conçus pour générer des analyseurs embarqués dans des serveurs. En effet, les serveurs Internet présentent des contraintes fortes en terme de gestion mémoire et de temps d'exécution. C'est pourquoi, les analyseurs embarqués dans les serveurs Internet sont la plupart du temps implantés à la main. Pire, la complexité d'analyseurs non bloquants est telle qu'ils sont systématiquement implantés de manière bloquante !

L'utilisation de *Tatoo*, un générateur d'analyseur syntaxique [23], m'a permis d'obtenir automatiquement des analyseurs de requêtes clientes [22]. Tatoo, contrairement aux principaux générateurs d'analyseurs actuels [36, 60, 90], permet de générer automatiquement des analyseurs qui sont compatibles avec des E/S bloquantes et non bloquantes. De plus, *Tatoo* a été conçu pour générer des analyseurs qui vont être embarqués dans des serveurs Internet et prend en considération les contraintes de temps d'exécution et de gestion mémoire induits.

Est-il possible de réaliser un logiciel de tests qui permet de simuler des coupures dans les requêtes avant de mettre en exergue les capacités des analyseurs syntaxiques non bloquants ?

Troisième partie

Performance et optimisation

6

Développement d'un serveur HTTP performant

Sommaire

Les services Internet sont devenus de plus en plus importants aussi bien pour les industriels que pour les particuliers. Parmi les services les plus populaires, les sites d'information en ligne sont sujets, en fonction de leurs contenus et de l'actualité, à de très fortes variations du nombre de visiteurs pouvant parfois atteindre des facteurs d'ordre 20 en quelques minutes [64]. Cet exemple illustre les propriétés de disponibilité, de montée en charge et de capacité à supporter longtemps des charges importantes des services Internet. Plus techniquement, les services Internet sont fournis par des *serveurs logiciels* qui vont tenter de satisfaire les demandes des différents clients. Le développement d'un serveur Internet doit donc, en outre, prendre en compte ce problème sans précédent qu'est le *support d'un grand nombre d'utilisateurs accédant simultanément à un seul service.*

Pratiquement, les différentes actions concurrentes effectuées sur un serveur Internet vont se traduire par des opérations d'E/S sur des interfaces réseaux et sur le disque dur ainsi que des calculs sur la machine d'hébergement. Afin d'entrelacer les traitements des différentes requêtes clientes concurrentes, on utilise traditionnellement des processus ou des processus légers. Cependant, cette approche implique un coût important en terme d'empreinte mémoire, en temps d'ordonnancement ou en nombre de bascules du processeur [1, 41, 85]. Une autre approche consiste à utiliser des E/S non bloquantes mais, ce type d'E/S est difficile à utiliser car il est nécessaire de gérer manuellement la sauvegarde des contextes [1, 11, 12, 41].

Dans ce chapitre, je vais commencer par quelques considérations techniques sur les E/S

bloquantes et non bloquantes du langage Java. Je présenterai ensuite les différentes mises en œuvre des sélecteurs mécanisme utilisé dans le cas d'E/S non bloquantes en Java. J'expliquerai notamment comment associer les sélecteurs et les processus légers au sein d'une application. Enfin, pour montrer la faisabilité et l'intérêt de mon approche, j'ai réalisé des tests de performance à l'aide de deux outils : (i) ApacheBench [5] et (ii) Httperf [45]. Ces outils permettent d'exhiber les performances « pures » des serveurs HTTP dont la mesure la plus percutante est le nombre de requêtes traitées en une seconde. Grâce à ces tests, je montre que l'embarquement d'un analyseur au sein d'un serveur HTTP ne nuit pas à ses performances.

6.1 Considérations techniques

Les E/S sont des composantes disponibles depuis les premières versions de Java. Le paquetage java.io, qui regroupe l'ensemble des classes de gestion des flots d'octets, est apparu à la version 1.0 du langage. Quant aux classes de gestion des flots de caractères, elles sont apparues à la version 1.1. Celles gérant les connexions réseaux (*socket*) sont présentes dans le paquetage java.net depuis la première version.

L'âge de ses composants n'est pas signe d'obsolescence mais illustre l'absence de prise en compte des problèmes de performance, les E/S non bloquantes notamment. C'est pourquoi et depuis la version 1.4, Java inclut une nouvelle architecture d'E/S et de connexions réseaux fondamentalement différente de l'architecture existante [104].

6.1.1 Description des E/S Java

En Java, les flots d'octets ne fournissent qu'une partie des fonctionnalités d'E/S nécessaires au développeur. Ainsi les flots d'octets permettent la lecture des données d'un flot afin de créer un tableau d'octets et inversement d'écrire dans un flot un tableau d'octets. Cependant, toutes ces opérations sont *bloquantes* ce qui n'est pas toujours souhaitable. Il est difficile de lire plus d'un octet sans utiliser de tampon mémoire. Le paquetage java.io ne fournit que deux possibilités : (i) utiliser le tampon fourni par les flots « bufferisés » ou (ii) programmer son propre tampon mémoire.

C'est pourquoi, les nouvelles E/S présentes dans le paquetage java.nio ont été proposées afin de répondre à ce problème. Pour y remédier, elles fournissent : (i) des tampons mémoires qui permettent une gestion plus simple et plus performante des manipulations de zones mémoires (ii) des accès aléatoires plus rapides et (iii) facilitent le « mappage » de données en mémoire [97]. Dans le but de transférer des données depuis ou en direction d'un tampon mémoire, cette nouvelle architecture va utiliser le concept de *canal* qui, va représenter une source ou une destination (un fichier, une connexion réseau ou une file) pour les données. Cependant, un canal va être différent d'un flot car les E/S peuvent être aussi bien bloquantes que non bloquantes. Cependant, seul les canaux héritant de l'interface SelectableChannel peuvent être utilisés dans les deux modes.

6.1.2 Les tampons de données

Le ramasse-miettes de Java déplace les objets en mémoire par des opérations de copie. Dans la nouvelle architecture d'E/S, les tampons de données peuvent être de deux types : (i) directs et (ii) non directs. Leur différence principale est leur gestion ou non par le ramasse-miettes. Les tampons de données directs sont alloués dans des zones mémoires qui ne sont pas gérées par le ramasse-miettes. *A contrario*, les tampons de données non directs sont alloués dans des zones mémoires contrôlées par le ramasse-miettes.

En pratique, les tampons directs nécessitent un temps d'allocation supérieur au tampon de données non directs pour des tampons pas trop grand (voir Fig. 6.1).

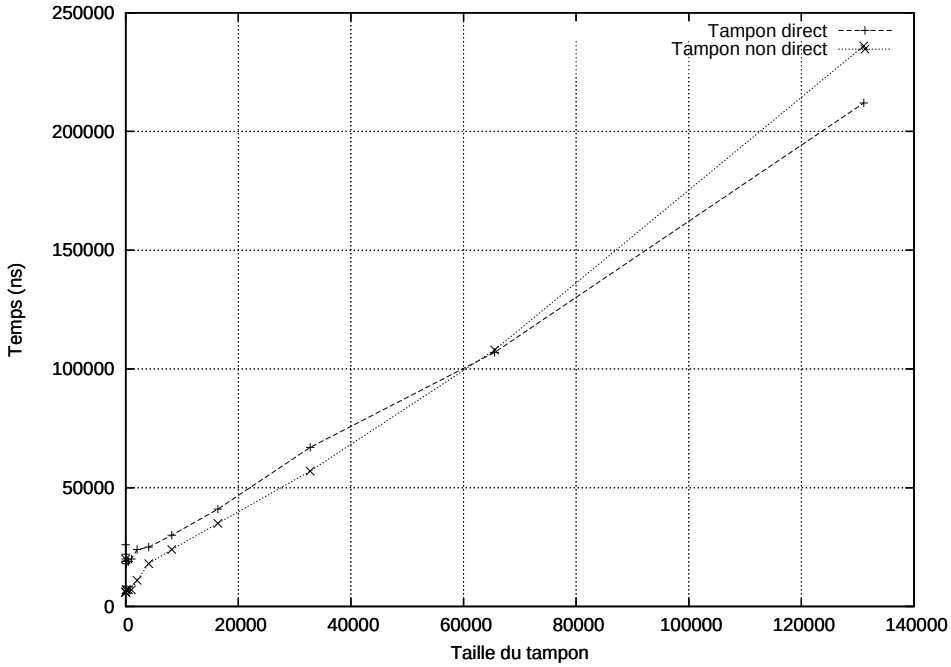


FIG. 6.1 – Temps d'allocation des tampons directs en non directs

En effet, dans le cas direct les blocs de mémoire sont gérés par une liste des blocs libres tandis que dans le cas direct ces blocs mémoires sont gérés par le ramasse-miettes. Cependant, à partir d'une certaine limite (65536 octets), les tampons non directs ne vont plus être alloués au même endroit dans la mémoire du fait de leur taille. Ils vont être positionnés dans des zones de mémoire très peu collectées par le ramasse-miettes (aucun déplacement). Les blocs de ces zones mémoires sont gérés par une liste des blocs libres comme dans le cas des tampons directs. Cependant, du fait d'une surcouche (le ramasse-miettes) les performances sont moins bonnes, ce qui explique le temps plus important de l'allocation des tampons non directs vis-à-vis des tampons directs.

Du fait des temps de création non négligeable, il est recommandé, lors du développement d'un serveur Internet, d'utiliser un vivier de tampons de données et ce dans les deux cas.

En résumé : Le contenu d'un tampon direct va résider dans une zone qui n'est pas gérée par le ramasse-miettes de la machine virtuelle Java. L'espace alloué pour ces tampons ne sera jamais recopié d'une zone à une autre, améliorant ainsi les performances d'une application à longue durée de vie.

Cependant, lors de tests pratiques d'un serveur HTTP en exploitation, les performances à l'utilisation de ces deux types de tampons sont identiques (voir Fig. 6.2).

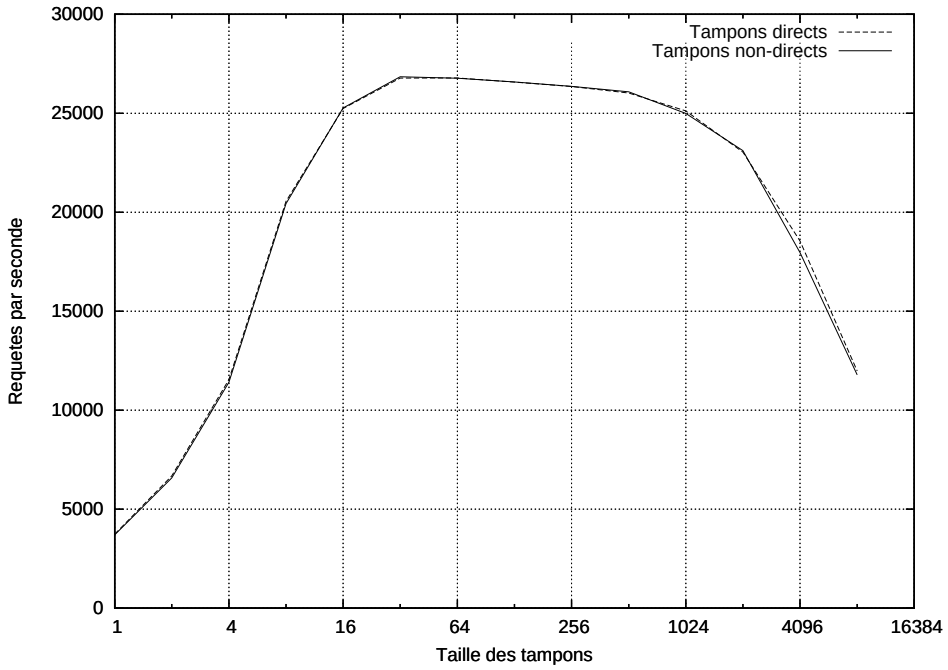


FIG. 6.2 – Performance des différents types de tampons de données dans un serveur HTTP

Lors d'une lecture-écriture les tampons non directs vont être recopiés temporairement sur la pile. A la fin d'une lecture, il va donc y avoir une recopie des données lues présentes dans le tampon non direct temporaire vers le tampon non direct que l'on utilise. Dans le cas direct, les appels travaillent directement sur le tampon. Cependant, les opérations de recopie sont des opérations directement optimisées au niveau système d'exploitation. Les copies sur la pile sont donc très efficaces. De plus, il semblerait que certaines optimisations ne soient pas présentes dans l'implantation des tampon directs actuels.

6.1.3 Les sélecteurs

Dans le modèle d'E/S non bloquantes, les applications sont capables de « chevaucher » le traitement des différentes E/S. Lors d'un appel à une procédure d'E/S, celui-ci va immédiatement retourner en indiquant que l'opération a été correctement initiée. L'application va pouvoir réaliser d'autres traitements tandis que l'opération d'E/S se poursuit en « arrière-plan ». Dès que des données sont disponibles sur l'interface, un signal est généré signalant à l'application qu'elle peut lire des données sur l'interface en question. Cependant, il est nécessaire de mettre en place des mécanismes pour récupérer le signal indiquant que des données

sont disponibles sur une interface. Ce *mécanisme de multiplexage* des E/S est mis en place à l'aide d'une procédure de sélection telle que *poll* sur System V ou *select* sur BSD Unix. Le mécanisme de sélection va surveiller l'activité de toutes les interfaces d'E/S (fichiers ou connexions réseaux). Si une activité se produit pour une interface d'E/S alors l'application va lire ou écrire les données ou une partie de celles-ci pour cette interface d'E/S.

***Clés* de sélection :** En Java, les interfaces d'E/S sont enregistrées auprès d'un sélecteur sous forme de *clés de sélection*. Pour chaque sélecteur, il y aura une association clé-interface d'E/S qui sera unique. La clé, qui est utilisée en interne par le mécanisme de sélection, permet aussi de spécifier les opérations que l'on souhaite effectuer sur une interface d'E/S (lecture, écriture, etc.). De plus, on pourra rajouter à cette association d'autres informations (tampons de données par exemple) dépendantes de l'application.

En théorie, le multiplexage d'E/S permet d'avoir un seul processus léger qui effectue tout le travail. En pratique, il n'est pas possible de masquer les temps de latence des E/S disque ou de tirer partie d'un système multi-processeurs. En effet, le temps de latence des E/S disque est dû à l'absence du support d'E/S non-bloquantes sur les fichiers par l'API NIO [104]. En effet, certains systèmes d'exploitation n'en fournissent pas le support [86]. Il est donc nécessaire d'incorporer le mécanisme des sélecteurs au sein d'applications possèdant plusieurs processus légers.

L'incorporation des sélecteurs au sein d'applications multi-processus peut être réalisée de différentes façons (dans les descriptions ci-dessous à un processus est associé un sélecteur) :

- 1 *sélecteur :* Cette architecture utilise un seul processus de sélection qui effectue tout le travail (acceptation, lecture, etc.).
- 1 *accepteur et* 1 *lecteur-écrivain :* Cette architecture utilise deux processus et donc deux sélecteurs. Le premier qui effectue l'acceptation des nouveaux clients et le second qui réalise les autres opérations (lecture, écriture, etc.). Il est important de noter que seule la *socket* serveur sera enregistrée dans le sélecteur d'acceptation. Les *sockets* clientes seront quant à elles enregistrées dans le second sélecteur.
- 1 *accepteur, N lecteurs-écrivains* : On utilise un seul processus d'acceptation et plusieurs processus qui réalisent les autres opérations (lecture, écriture, etc.). Seule la *socket* serveur sera enregistrée dans le sélecteur d'acceptation. De plus, il est nécessaire de mettre en place un mécanisme de répartition du travail entre les différents processus de traitements (tourniquet par exemple).
- 1 *accepteur, N lecteurs et M écrivains* : On utilise un processus d'acceptation et plusieurs processus de traitements. Les processus de traitements vont être divisés équitablement ou non en lecteurs et écrivains. Les processus lecteurs vont enregistrer les connexions clientes pour les opérations de lecture et les processus écrivains pour les opérations d'écriture.

Dans les cas présentés ci-dessus, les connexions entrantes sont traitées par un processus d'acceptation. Il doit aussi distribuer les événements sur les autres processus de sélections. Cependant, dans la première architecture, tous les événements d'E/S sont traités par le

même processus de sélection. Dans les autres cas, le traitement va être délégué à d'autres processus de sélection.

Chacune des architectures données ci-dessus va présenter des avantages et des inconvénients résumés par le tableau suivant :

Architecture	Avantages	Inconvénients
1 sélecteur	Le plus simple à coder.	Problème des E/S non bloquantes sur le disque dur.
1 accepteur et 1 lecteur-écrivain	Simple à coder. Limitation facile sur le nombre de clients actifs (sélectionnés).	Problème des E/S non bloquantes sur le disque dur.
1 accepteur et N lecteurs-écrivains	Performant. Distribution de la charge sur plusieurs processus. Passage d'information simple. Performant.	Dur à coder.
1 accepteur, N lecteurs et M écrivains	Très performant. Bonne gestion de la charge lecteurs-écrivains.	Extrêmement dur à coder. Risque d'interblocages. Passage d'informations compliquées.

6.1.3.1 Architecture 1 accepteur et 1 lecteur-écrivain

Cette architecture utilise deux processus et donc deux sélecteurs. Le premier qui effectue l'acceptation des nouveaux clients et le second qui réalise les autres opérations (lecture, écriture, etc.). Il est important de noter que seule la *socket* serveur sera enregistrée dans le sélecteur d'acceptation. Les *sockets* clientes seront quant à elles enregistrées dans le second sélecteur. Les deux processus vont utiliser une file bloquante (LinkedBlockingQueue) fournie par l'API Java pour communiquer entre eux. Cette architecture doit faire face à deux problèmes :

1. Minimiser le nombre de clés dans le sélecteur lecteur-écrivain.
2. Supprimer les clients qui sont lents.

Le second problème permet de meilleure performance du serveur et répond partiellement au premier problème.

Minimisation du nombre de clés dans le sélecteur

Le problème de minimisation du nombre de clés au sein d'un sélecteur est lié à l'existence d'une borne physique des mécanismes de sélection. Cette borne est directement lié au système d'exploitation sous jacent. Au dela de ce seuil les performances de l'opération de sélection vont chuter. La première solution est de mettre en attente les clients lorsque cette borne est atteinte au niveau du sélecteur. Cependant cette solution n'est pas acceptable en terme de performance. Une autre solution est de répartir la charge en utilisant plusieurs processus lecteurs-écrivains qui est l'architecture 1 accepteur et N lecteurs-écrivains (voir 6.1.3.2).

Suppression des clients lents

Les clés de sélection vont représenter des connexions ouvertes entre le serveur et ses clients et donc des objets utilisés (tampons mémoires, analyseurs syntaxiques, etc.). Dans le cas de clients très lents ou déconnectés intempestivement le serveur utilisera des objets « inutilement ». Pour optimiser l'utilisation des objets et donc optimiser la consommation mémoire, il est particulièrement intéressant de mettre en place des *quantums* de temps sur les clés. Ainsi, si une clé n'a jamais été sélectionnée durant cette période, la connexion peut être fermée et le client devra se reconnecter. De plus supprimer les clients qui sont lents permet de minimiser le nombre de clés présentes dans un sélecteur et répondre, du moins partiellement, au problème précédent.

Comment mettre en œuvre pratiquement la suppression des clients lents ?

Saburo fournit une classe d'encapsulation permettant de gérer plus facilement les sélecteurs fournis par l'API NIO. De plus, cette classe permet de prendre en compte deux niveaux de sélection possible : (i) sélection sur une file d'événements (méthode performsEvent()) et (ii) sélection d'événements d'E/S (méthode performsIo(SelectionKey key)). Le mécanisme de sélection est le suivant :

```
public void doSelect() throws SelectionException {
  int numUpdateReadyKeys = selector.select(timeout) ;

  performsEvent() ;

  for(Iterator< SelectionKey > it = selector.selectedKeys().iterator() ; it.hasNext() ; ) {
    SelectionKey key = it.next() ;
    performsIo(key) ;
    it.remove() ;
  }
  postSelect() ;
}
```

La méthode postSelect() permet de mettre en place des mécanismes d'expiration des clés de sélection. Si une clé n'a pas été sélectionnée pendant un *quantum* de temps donné, elle sera supprimée de l'ensemble des clés de sélection. Par exemple cette méthode d'expiration peut

être :

```
public void expire(Iterator< SelectionKey > keys) {
  if(timeout <= 0L) {
    return ;
  }

  long currentTime = System.currentTimeMillis() ;
  if(currentTime < nextKeysExpiration) {
    return ;
  }

  nextKeysExpiration = currentTime + timeout ;

  for( ; keys.hasNext() ; ) {
    SelectionKey key = keys.next() ;
    if(key.isValid()) {
      long expire = (Long)key.attachment() ;
      if(currentTime - expire >= timeout) {
        cancel(key) ;
      } else if(expire + timeout < nextKeysExpiration) {
        nextKeysExpiration = expire + timeout ;
      }
    }
  }
}
```

Ce mécanisme est très intéressant pour optimiser les performances d'un serveur HTTP car il permet de rejeter les clients trop lents, les obligeant alors à se reconnecter au serveur. De plus, le serveur ne maintiendra pas les clients qui sont déconnectés intenpestivement.

6.1.3.2 Architecture 1 accepteur et N lecteurs-écrivains

La façon la plus simple de répondre aux problèmes de blocage de tout le serveur lors d'accès à une base de données ou au disques est d'utiliser un processus d'acceptation et plusieurs de traitements. Lorsqu'un client souhaite se connecter au serveur, le processus d'acceptation l'accepte et établit une connexion. Il transmet ensuite cette connexion à un processus de traitement. Il n'est pas nécessaire ici de prendre soin d'annuler toutes clés ayant trait à cette connexion dans le sélecteur d'acceptation car il n'y a que la *socket* serveur d'enregistrée dans celui-ci. De fait il n'est pas nécessaire de s'assurer de la synchronisation des clés ou des sélecteurs. Le sélecteur présent dans un processus aura donc le comportement suivant :

1. se bloquer sur le `select()` ;
2. en sortir seulement s'il y a des opérations prêtes à être exécutées ;
3. exécuter toutes les opérations qui sont prêtes (acceptation dans un cas et lecture/écriture dans l'autre) ;

4. revenir au select.

Dans ce type d'architecture qui correspond à l'architecture MSPED, il est nécessaire d'assurer une bonne distribution du travail entre les threads, j'utilise un simple algorithme de *round-robin* pour assigner les connexions entrantes aux processus de traitements. Ceci fonctionne plutôt bien en pratique mais dans certaines situations, il peut être nécessaire de prendre en considération d'autres facteurs tels que l'activité de chaque processus pour assurer une meilleure distribution des connexions pour limiter le nombre de clés présentes dans un sélecteur.

6.1.3.3 Architecture M lecteurs et N écrivains

Cette architecture est très complexe dans l'utilisation des synchronisations. En effet, il y a beaucoup de points d'interaction entre les processus de sélections lecteurs et écrivains. Ainsi, les étapes 1 et 2 présentées précédemment peuvent survenir en même temps, ce qui crée de nombreux problèmes de concurrence. Les effectuer dans le même processus évite beaucoup de problèmes dus au accès concurrent sur une même ressource. De plus, le couple sélecteur et clés de sélection associées à celui-ci n'est pas protégé contre des accès concurrents. Si une clé de sélection est modifiée par un processus pendant qu'un autre processus appelle la méthode *select(...)* de son sélecteur, une exception est levée. Ce phénomène arrive souvent lorsque les processus consommateurs ferment les canaux et annulent ainsi indirectement les clés de sélection correspondantes. Si une méthode *select(...)* est appelée à ce moment, la clé de sélection sera annulée de façon impromptue et l'exception CancelledKeyException sera levée. Il en résulte la règle de simplification suivante :

Un sélecteur, ses clés de sélection et les canaux enregistrés ne doivent jamais être accessibles par plus d'un processus à la fois.

Cependant, il est nécessaire de passer outre cette règle de simplification en répondant aux problèmes suivants :

1. Communication entre les processus lecteurs et écrivains.
2. Fermeture des connexions.

Communication entre les processus lecteurs et écrivains

La communication entre les processus lecteurs et écrivains se fait *via* des files bloquantes (LinkedBlockingQueue) fournie par l'API Java. Ces files gérent de manière efficace les synchronisations. C'est pourquoi, un processus lecteur ou écrivain devra « sélectionner » un événement sur cette file et réaliser une sélection sur les événements d'E/S. Cependant l'API fournie par le paquetage java.nio ne permet pas de réaliser ces sélections qui s'opèrent à des niveaux différents. En effet, les sélections sur la file d'événements sont faites sur des données logiques tandis que la sélection sur les événements d'E/S est une sélection sur des données physiques. Il est nécessaire de fournir un sélecteur plus évolué que celui fournit par l'API pour permettre cette sélection sur des files et sur les E/S.

Comment réaliser une sélection sur une file et sur des descripteurs ?

Un sélecteur va être constitué d'une file des événements en entrée et d'un sélecteur d'E/S de l'API nio. Le processus réalisant l'insertion dans une file va réaliser une opération push bloquante dans la file. Si la file est pleine, l'opération va bloquer tant qu'au moins un élément de la file n'a pas été retiré. La suppression d'un élément de la file va être une opération non bloquante ce qui permet de bloquer le processus consommateur sur la sélection des E/S.

Mais comment peut on savoir qu'un événement est disponible dans la file ?

Pour le savoir, l'opération push va réveiller le sélecteur d'E/S. Le processus de sélection va alors retirer l'événement dans la file puis enregistrer des informations portées par cet événement pour une sélection d'E/S.

J'ai implanté un autre mécanisme qui utilise la classe Pipe de l'API java.nio. Une file va être représentée par l'association d'une file « classique » et d'un Pipe. Lorsqu'un événement est mis dans la file, un octet physique va être écrit dans le Pipe. Comme le sélecteur d'E/S réalise une sélection sur ce Pipe, il sera réveillé et le processus pourra récupérer la main.

Ce mécanisme est très similaire au mécanisme qui utilise la méthode wakeup. En effet, l'implantation de cette méthode utilise une clé dédiée au réveil du sélecteur. Lorsque l'on fait appel à la méthode wakeup, cette clé est positionnée comme étant sélectionnable, le sélecteur est donc réveillée et le processus pourra reprendre la main. Cependant, l'opération wakeup est une opération fournie en interne par l'API. Elle bénéficie donc d'optimisations qui ne sont pas disponibles lorsque l'on implante le mécanisme à la main à l'aide de la classe Pipe.

Fermeture des connexions

Les connexions réseau clientes doivent pouvoir être fermées de manière asynchrone pour prendre en compte le problème d'enregistrement d'une même connexion dans plusieurs sélecteurs. La fermeture des connexions est basée sur un mécanisme asynchrone de fermeture des connexions et donc d'annulation des clés d'un sélecteur. Ainsi, il est nécessaire de ne pas annuler directement une clé mais plutôt d'annuler une opération possible sur une clé. Si la connexion est fermée en écriture on informe qu'il n'est plus possible d'écrire dans la connexion (shutdownOutput()). La connexion sera réellement fermée quand les deux types possibles d'opération seront annulées sur la connexion. Une fois les deux fermetures reçues, la clé pourra alors être annulée.

Pratiquement, j'utilise deux vues sur une même connexion. Ces vues vont servir de décorateur [37] à la classe SaburoSocket qui elle-même permet de simplifier l'utilisation de la classe SocketChannel de l'API NIO en restreignant certaines méthodes. La première vue est utilisée pour la lecture (RequestStream) et la seconde pour l'écriture (ResponseStream). Le développeur utilise l'une ou l'autre des vues selon ses besoins. La classe RequestStream ne permet que de lire des données sur la connexion réseau, tandis que la classe ResponseStream ne permet que d'écrire des données. Lorsque l'on ferme l'une ou l'autre des vues, un « signal » est envoyé *via* la

méthode shutdownInput (respectivement shutdownOutout) qui indique que la connexion réseau ne peut plus lire (respectivement écrire). Si les deux vues sont fermées alors la connexion réseau sera effectivement fermée.

6.2 Choix du protocole HTTP pour les tests

L'intérêt d'étudier les serveurs HTTP est l'existence d'une très grande variété d'outils permettant de mesurer leurs performances [5, 45] et d'autres qui sont commerciaux [99]. Le grand nombre de résultats qui sont publiés et qui comparent les performances de différentes architectures et/ou serveurs HTTP.

Pour comparer le serveur HTTP développé avec *Saburo* et *Tatoo*, je vais prendre en considération le serveur Apache httpd [7] ainsi que 3 autres serveurs Grizzly 1.7 [102], Jetty 6.0 [81] et Tomcat NIO 6.0 [107] qui sont développés en Java et qui utilisent l'API NIO [104]. Dans cette section, je vais présenter succinctement ces serveurs.

6.2.1 Apache httpd

Apache httpd [7] utilise un vivier de processus de traitements dont la taille varie dynamiquement en fonction du nombre de connexions clientes concurrentes. Chaque processus traite un seul client et réalise l'ensemble des traitements nécessaires pour répondre à sa requête. La taille du vivier de processus va donc limiter le nombre de connexions qui sont acceptées simultanément par le serveur. Apache httpd est implanté en C.

6.2.2 Grizzly 1.7

Grizzly [102] est un connecteur HTTP basé sur l'API NIO. Initialement, Grizzly a été conçu pour travailler au dessus de Coyote le connecteur HTTP d'Apache Tomcat. Cependant et comme beaucoup d'autre connecteur HTTP, sa montée en charge est limitée par le nombre de processus disponible et lorsque l'on utilise des connexions persistantes, il souffre énormément du paradigme d'*un processus par connexion*. C'est pourquoi, la montée en charge est la plupart du temps limité par le nombre de processus que peut supporter la plateforme.

Avec l'intégration des NIO dans le JDK 1.4, Grizzly a été initié comme un projet indépendant dont le but était de fournir les meilleurs performances possibles pour un serveur HTTP et d'illustrer la faisabilité d'utiliser les NIO pour ce même protocole. Grizzly diffère de Coyote car il permet de configurer tous les viviers de processus et supporte surtout les E/S non bloquantes.

Pour détecter la fin d'une requête, Grizzly utilise une machine à états finie qui analyse l'information d'en-tête de la longueur du message, ce qui permet de prédire la fin du flot. L'acceptation des nouveaux clients est réalisée dans un processus dédié et les autres lectures/écritures utilisent un vivier de processus. Les sélecteurs sont à temps borné ce qui permet de supprimer les requêtes longues et d'améliorer les performances du serveur. Enfin,

Grizzly cherche toujours à bloquer au minimum les processus, c'est-à-dire qu'il va tenter, par exemple, d'écrire une donnée dès que celle-ci sera initiée. Si l'opération n'a pas réussi ou n'est pas complète, alors c'est seulement à ce moment-là qu'elle sera enregistrée au sein d'un sélecteur.

Dans le cas des connexions persistantes, le processus de traitement attribué à une requête sera relâché et la connexion sera enregistrée en vue d'une nouvelle opération de lecture. Inversement, dans le cas de connexion non persistante le processus de traitement et la connexion seront relâchés dans les viviers. Cette stratégie permet d'éviter le problème du paradigme d'*un processus par requête* en exploitant la nature sans état du protocole HTTP. Il est ainsi possible de traiter quelques milliers de clients en utilisant seulement quelques dizaines de processus.

6.2.3 Jetty 6

Jetty[81] est un serveur HTTP pour des pages statiques et dynamiques. Contrairement à des solutions séparant le serveur du « conteneur », Jetty embarque le serveur HTTP et l'application HTTP (les servlets) ce qui permet d'éviter des surcoûts dus aux inter-connexions et des complications d'implantation.

Jetty 6 est une nouvelle implantation du serveur HTTP Jetty et du conteneur de servlet. Contrairement à l'ancienne version, il y a une réduction des dépendances avec d'autres logiciels et une intégration de différentes technologies dont les NIO.

De nouveaux connecteurs sont fournis qui se basent sur les E/S non bloquantes de Java. Cette bibliothèque, comme je l'ai déjà souligné auparavant permet d'intégrer des E/S non bloquantes et des processus de traitements peuvent être attribués aux requêtes réellement actives. Ainsi quand une connexion n'est plus active, le processus peut être rendu au vivier de processus et la connexion est enregistrée au sein d'un sélecteur pour détecter que de nouvelles données arrivent. Ce modèle permet de fournir une montée en charge plus importante du serveur car le serveur traitera plus de clients qu'une version basée sur les processus légers.

6.3 Tests de performance

Cette section présente une évaluation des performances de *Banzai*, mon server HTTP développé précédemment. Dans *Banzai*, je me suis particulièrement focalisé sur les aspects de (i) robustesse face à la charge et de (ii) très forte concurrence. Mon but est ici d'évaluer la faisabilité d'intégrer un analyseur syntaxique au sein d'un serveur HTTP sous des conditions de charge extrêmes. Les serveurs HTTP forment l'archétype des applications qui doivent :

- monter fortement en charge ;
- être robustes face aux variations très importantes dans le temps de celles-ci.

Le principal intérêt d'étudier les serveurs HTTP est la grande variété d'outils de tests (commerciaux ou non) pour mesurer leur performance et du grand nombre de résultats déjà

publiés.

Dans cette section, je vais présenter l'environnement de tests, ainsi que les deux outils libres utilisés (i) ApacheBench [5] et (ii) Httperf [45]. ApacheBench est un standard de test *de facto*. Il existe d'autres standards mais ils sont commerciaux [99]. Je vais comparer les performances de *Banzai* avec celles obtenues pour les quatre autres serveurs décrits précédemment. Ces résultats montrent que l'intégration d'un analyseur syntaxique ne nuit en aucun cas aux performances du serveur HTTP. De plus et bien que *Banzai* ne présente pas toutes les fonctionnalités des autres serveurs (conteneur de servlet, SSL, etc.) les tests sont effectués dans le contexte particulier de charge importante. Ce qui n'entraîne aucune pénalité comparativement aux autres serveurs.

Remarque : Dans les tests suivants, j'utilise seulement un serveur Internet basé sur le modèle de concurrence SEDA pour les tests (nommé *Banzai*). En effet, *Banzai* bénéficie d'un certain nombre d'optimisations qui n'ont pas été intégrées dans tous les générateurs de Saburo. Par exemple, lors de l'écriture de la réponse, *Banzai* va tenter de l'écrire sans passer par le mécanisme de sélection. Si l'écriture n'est pas possible ou n'est pas finie alors la connexion dans laquelle on souhaite écrire sera enregistrée dans un sélecteur pour cette opération d'écriture. L'intérêt de cette optimisation est d'éviter de passer par un sélecteur alors que l'opération est possible. Il y a donc un gain de temps car la sélection n'est pas effectuée.

6.3.1 Environnement de tests

Toutes les mesures de performances sont effectuées sur une machine Pentium IV bi-processeur 2.4GHz avec 2GB de mémoire vive. Le système d'exploitation sous-jacent est une distribution Gentoo avec un noyau Linux 2.6.19. La machine virtuelle Java est la machine virtuelle de Sun, HotSpot version 1.7.0-ea-b24 (correction de « bugs » dans les sélecteurs). Le client a exactement la même configuration. Les deux machines sont connectées *via* un aiguilleur ethernet Gigabit. Bien que cette configuration ne simule en aucun cas les effets d'un vrai réseau, mon interêt principal est de montrer les performances et la stabilité du serveur sous une charge très importante. Tous les tests sont effectués pour des pages Internet statiques dont la taille varie de 4K à 1M.

6.3.2 ApacheBench

6.3.2.1 Description

ApacheBench [5] est un petit outil qui permet de mesurer les performances de n'importe quel serveur HTTP. Il a été conçu pour donner une idée des performances qu'un serveur HTTP et ses différentes configurations peuvent fournir. En particulier, il montre combien de requêtes peuvent être traitées par seconde par le serveur, ainsi que la bande passante et le temps moyen de traitement d'une requête.

Cet outil est directement intégré comme module du serveur web Apache httpd. Plus d'informations concernant les options de cet outil sont données dans [5]

6.3.2.2 Variation du nombre de clients

Ce test permet de comparer les performances des différents serveurs en fonction du nombre de client connectés en même temps au serveur. La version du protocole HTTP utilisée ici est la version 1.0. Le fichier qui est demandé par les clients est statique et a une taille de 4096 octets.

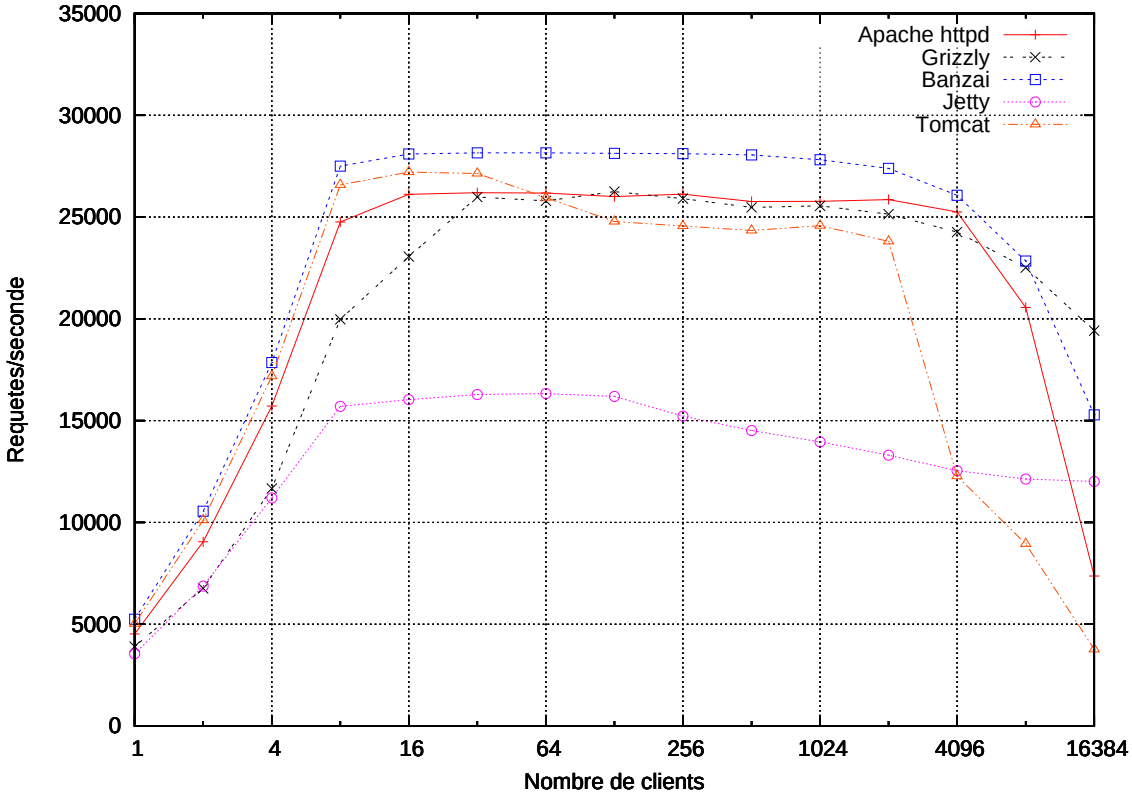


FIG. 6.3 – Variation du nombre de requêtes par seconde en fonction du nombre de clients.

On remarque globalement que pour mon serveur HTTP *Banzai* et l'ensemble des serveurs de comparaison, il y a une augmentation des performances jusqu'à 16 clients connectés en même temps. Les performances stagnent ensuite jusqu'à environ 2048 clients connectés en même temps. Tous les serveurs subissent ensuite une chute de leur performance. Pour les serveurs utilisant des E/S non bloquantes dont *Banzai*, cette chute provient d'une limitation système sur le nombre de descripteurs de fichiers ouvert en même temps sur le serveur. L'algorithme de gestion des descripteurs libres semblent perdre en performance à partir de 4096 fichiers ouverts en même temps (bien que le serveur puisse en gérer 65536 au maximum).

Concernant le serveur httpd, qui utilise des processus légers, ses performances chutent du fait du trop grand nombre de processus légers au niveau de l'ordonnanceur et de la consom-

mation mémoire qui devient trop importante.

Concernant les performances du serveur Jetty, celles-ci sont très en deçà des performances des autres serveurs non bloquants. Il semblerait que cela soit dû à une mauvaise configuration du serveur. Je n'ai pas réussi à résoudre ce problème n'ayant eu aucun retour, malgrè de nombreuses demandes aux développeurs de ce serveur. *Banzaï*, passé le seuil de 8192 clients, a une chute importante des performances car, contrairement aux autres serveurs, il maintient et ouvre toujours les connexions comme persistantes.

6.3.2.3 Variation de la taille du fichier

Ce test permet de comparer les performances des différents serveurs en fonction de la taille du fichier envoyé aux clients. La version du protocole HTTP utilisée est là aussi la version 1.0. Le nombre de clients connectés en même temps au serveur est de 1024.

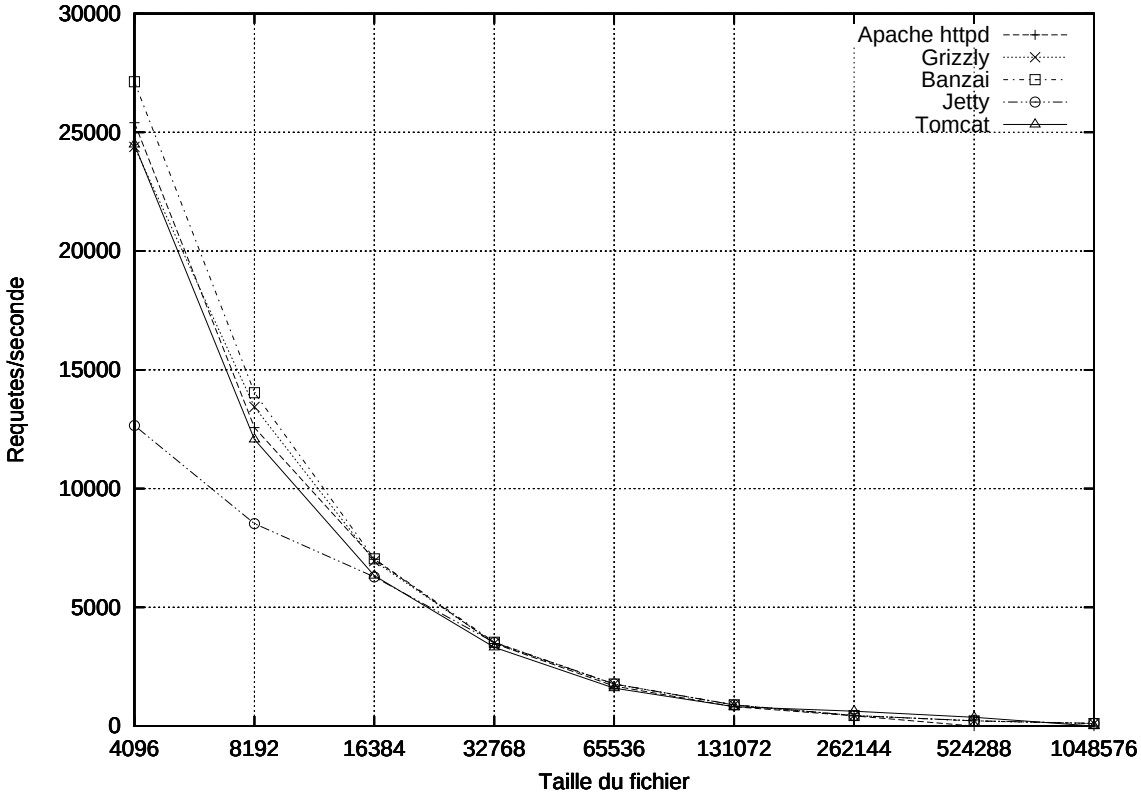


FIG. 6.4 – Variation du nombre de requêtes par seconde en fonction de la taille du fichier.

Globalement, on remarque que l'ensemble des performances des serveurs dont mon serveur HTTP *Banzai* suit exactement la même évolution. Ce phénomène vient du fait que plus on lit ou écrit d'informations dans une connexion réseau ou sur un disque dur et plus le système va prendre de temps pour le faire. Le test suivant (*microbenchmark*) permet d'illustrer ce phénomène. Plus les fichiers que l'on écrits sont gros et plus on prend de temps pour les écrire (voir Fig. 6.5). Intuitivement, la fonction entre le volume des données à écrire et le temps pris pour les écrire est de la forme : $f(x) = 1/x$

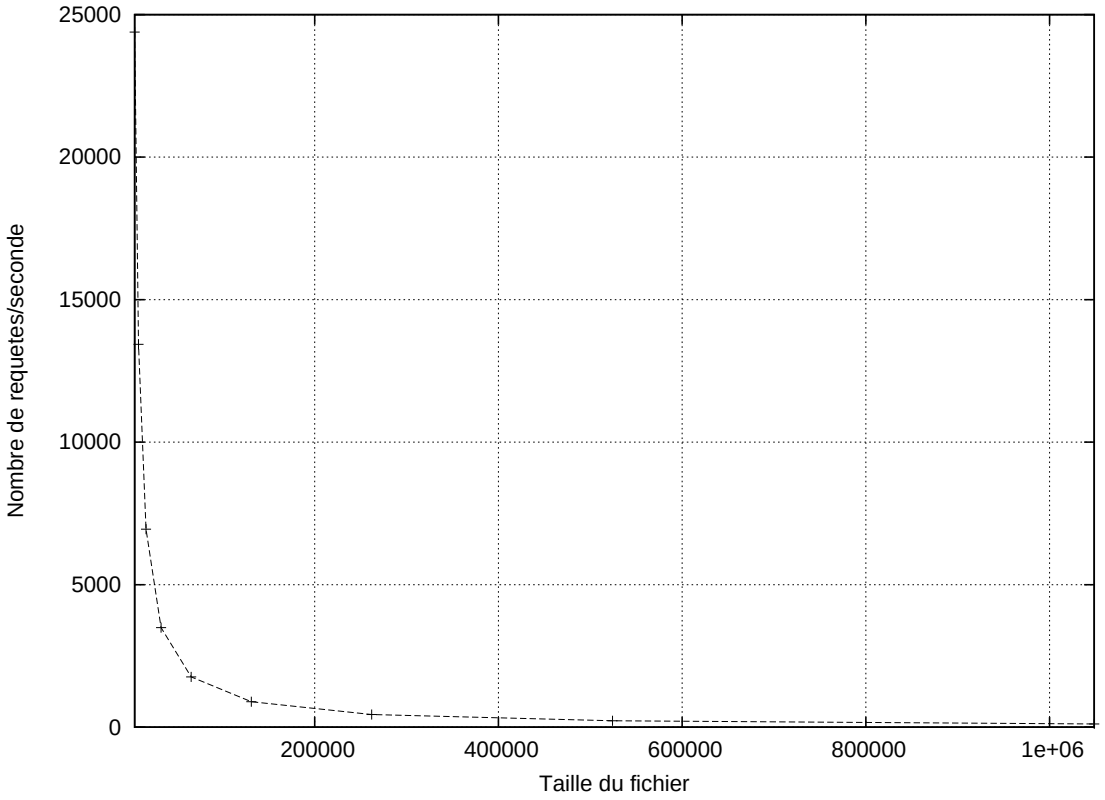


FIG. 6.5 – Variation des performances en fonction de la taille du fichier

Ce phénomène va complètement « écraser » les optimisations que l'on peut faire au niveau d'un serveur.

6.3.3 Httperf

6.3.3.1 Description

Httperf [45] est un outil qui permet de mesurer les performances des serveurs Internet. Il fournit différentes méthodes de flot de requêtes. Le but d'Httperf n'est pas d'implanter un protocole de tests particulier mais bien de fournir un outil robuste et performant pour faciliter la construction de protocole de tests. Les trois principales caractéristiques d'httperf sont :

1. la robustesse ;
2. le support du protocole HTTP/1.1 ;
3. l'adaptabilité pour ajouter de nouveaux protocoles de tests et de mesures de performance.

Ma principale motivation d'utiliser Httperf est le support de HTTP/1.1 ce qui n'est pas le cas d'ApacheBench. En effet et contrairement à HTTP/1.0 où la persistance est spécifiée

dans une ligne d'en-tête qui peut-être potentiellement la dernière, le protocole HTTP/1.1 considère automatiquement les connexions comme persistantes. Ainsi en HTTP/1.1 et dans le cas de la méthode GET, on peut répondre dès la fin de l'analyse de la première ligne de la requête. Ce qui améliore les performances du serveur.

Plus d'informations concernant les options de cet outil sont données dans [45].

6.3.3.2 Variation du nombre de clients

Ce test permet de comparer les performances des différents serveurs en fonction du nombre de clients connectés en même temps au serveur. La version du protocole HTTP utilisée ici est la version 1.1. Le fichier qui est demandé par les clients est statique et a une taille de 4096 octets.

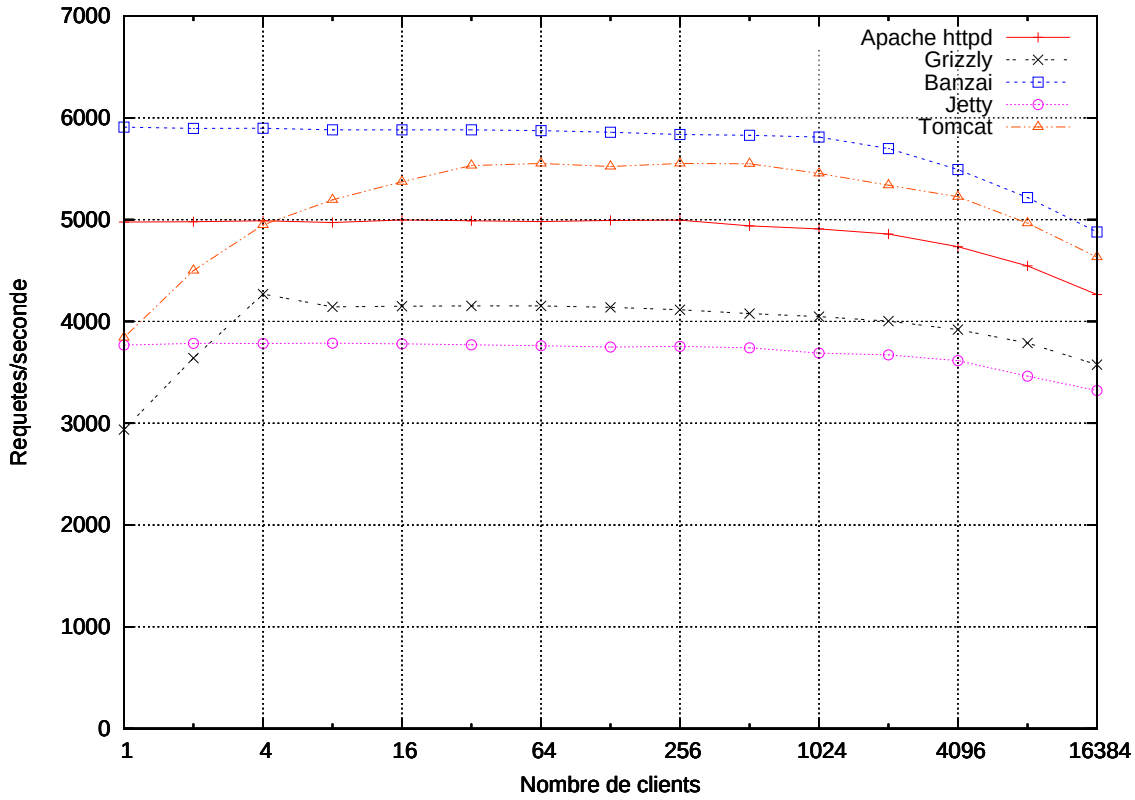


FIG. 6.6 – Variation du nombre de requêtes par seconde en fonction du nombre de clients.

Globalement, on remarque que pour l'ensemble des serveurs dont mon serveur HTTP *Banzai*, les performances sont relativement stables quelque soit le nombre de clients. L'ensemble des serveurs subissent là aussi une chute de leur performance pour environ 4096 clients. Dans ce test, on voit que *Banzaï* exploite parfaitement les caractéristiques du protocole HTTP/1.1 contrairement aux autres serveurs. Notament, le fait que toutes les connexions sont persistantes et qu'il n'est donc pas nécessaire d'analyser toute la requête cliente. Jetty contrairement au test avec ApacheBench présente ici de meilleures performances. Avec un fichier de configuration réellement dédié aux performances, les résultats pour ce serveur seraient

nettement plus probant. Grizzly présente des performances très en deçà de celles visibles lors du test avec ApacheBench. Je pense que cela vient du fait de l'utilisation du protocole HTTP/1.1 et de l'analyse bloquante des requêtes. Enfin, httpd présente de très bonnes performances, stables avec une chute à partir de 1024 clients due au coût de l'ordonnancement et de la consommation mémoire induite par l'utilisation des processus légers.

6.3.3.3 Variation de la taille du fichier

Ce test permet de comparer les performances des différents serveurs en fonction de la taille du fichier envoyé aux clients. La version du protocole HTTP utilisée est là aussi la version 1.1. Le nombre de clients connectés en même temps au serveur est de 1024.

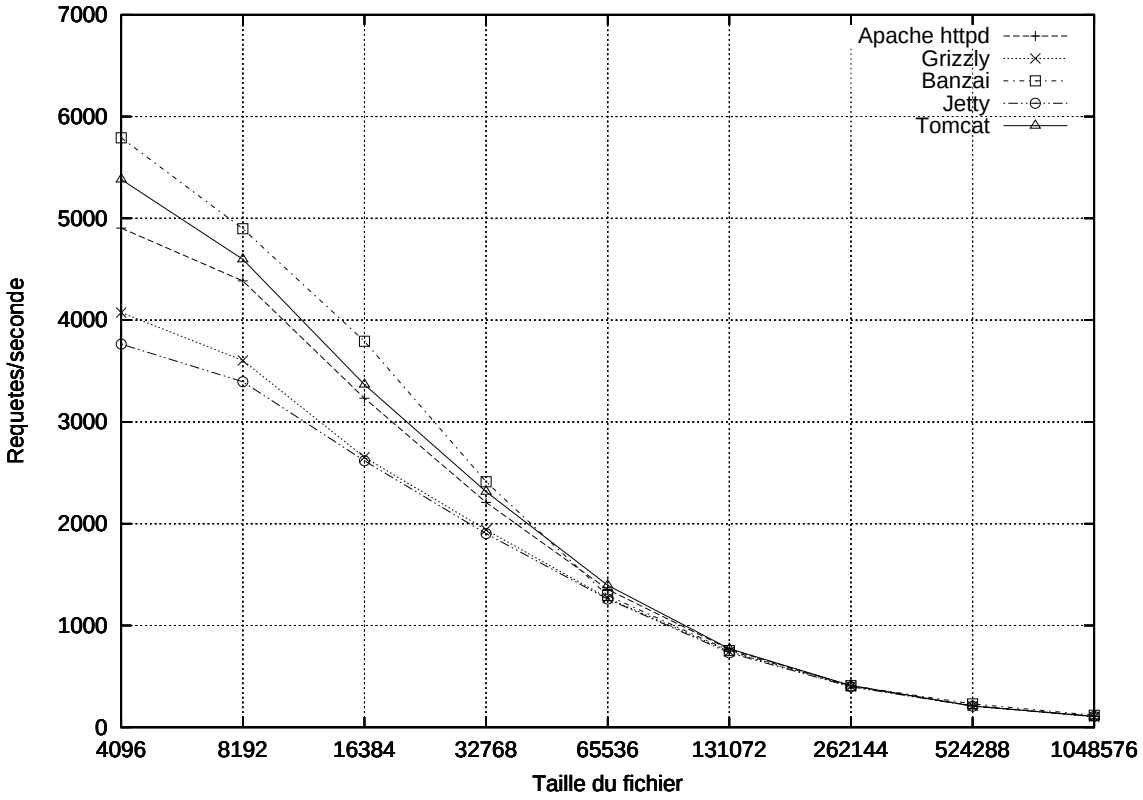


FIG. 6.7 – Variation du nombre de requêtes par seconde en fonction de la taille du fichier.

On remarque que globalement, l'ensemble des performances des serveurs dont mon serveur HTTP *Banzai* suit exactement la même évolution. Ce phénomène vient du fait que plus on lit ou écrit d'informations dans une connexion réseau ou sur un disque dur et plus le système va prendre de temps pour le faire. Contrairement au test avec ApacheBench, les performances vont « s'écraser » plus tardivement. Le protocole $HTTP/1.1$ a été introduit afin d'améliorer globalement les performances des serveurs Internet.

6.4 En conclusion...

Obtenir des serveurs Internet performant requiert un long travail d'optimisation. Ces optimisations nuisent à la clarté du code et pouvoir les générer automatiquement sans modifier le code métier d'un serveur Internet est particulièrement intéressant. Les mécanismes d'E/S et de concurrence fournis par le langage Java peuvent être particulièrement ardus à mettre en œuvre et à combiner. L'utilisation de Saburo permet dans un premier temps de masquer ces mécanismes difficiles d'utilisation et en les générant, évite de nombreux comportements non désirés (interblocages). De plus dans le cas du modèle de concurrence SEDA, Saburo fournit d'importantes optimisations qui permettent d'obtenir un serveur HTTP performant (nommé *Banzaï*). Ces performances illustrent parfaitement la faisabilité de l'embarquement d'un analyseur syntaxique au sein d'un serveur HTTP performant.

J'ai réalisé des tests de performance à l'aide de deux outils : (i) ApacheBench [5] et (ii) Httperf [45]. Ces outils permettent principalement de mettre en avant les performances « pures » des serveurs dont la mesure la plus percutante est le nombre de requêtes traitées en une seconde. Cependant, dans des conditions réelles d'utilisation, les requêtes n'arrivent pas nécessairement les unes à la suite des autres. De plus, certaines requêtes peuvent être découpées. Ces deux logiciels de tests ne permettent pas de modéliser ce type de comportement et l'intérêt d'utiliser des analyseurs non bloquants est justement de ne pas bloquer un processus sur la phase d'analyse et de pouvoir traiter une autre requête en attendant la suite de la ou des requête(s) en cours.

7

Conclusion et perspectives

Les services Internet sont devenus de plus en plus importants aussi bien pour les industriels que pour les particuliers. Ces services sont fournis par des *serveurs logiciels* qui tentent de satisfaire simultanément les demandes de très nombreux clients. C'est pourquoi le développement d'un serveur Internet doit tenir compte du *support simultané d'un grand nombre d'utilisateurs*. En plus de cette concurrence massive, le développement d'un serveur Internet doit tenter de résoudre les problèmes : (i) de *robustesse face à la charge*, (ii) d'*hébergement sur des machines hétérogènes*, (iii) de *génération de contenu dynamique* ou enfin, (iv) d'*évolution des fonctionnalités fournies*.

7.1 Rappel de la problématique

Techniquement, les différentes actions concurrentes effectuées sur un serveur Internet vont se traduire par des opérations d'E/S sur des interfaces réseaux et sur le disque dur ainsi que des calculs sur la machine d'hébergement. Afin d'entrelacer les traitements des différentes requêtes clientes, on utilise traditionnellement des processus ou des processus légers. Cependant, cette approche implique un coût important en terme d'empreinte mémoire, en temps d'ordonnancement ou en nombre de bascules du processeur entre les différents processus à exécuter [1, 41, 85]. Une autre approche consiste à utiliser des E/S non bloquantes mais, ce type d'E/S est extrêmement difficile à utiliser car il est nécessaire de gérer manuellement la sauvegarde des contextes [1, 11, 12, 41].

Il existe actuellement plusieurs architectures matérielles possibles : (i) machines mono-processeur, (ii) machines multi-processeurs, (iii) clusters, etc. ainsi que plusieurs *modèles de concurrence*. Les *modèles de concurrence* sont les techniques pour mettre en œuvre la concurrence au sein de ces applications. Je les ai caractérisés selon (i) l'utilisation des processus et (ii) le type d'E/S utilisé. De plus, lors du développement d'une application le modèle de concurrence va fortement structurer le code de celle-ci. Pratiquement, les modèles de concurrence vont avoir des coûts plus ou moins importants en termes de ressources. Ainsi, un serveur Internet peut être utilisé pour configurer un pda *via* une connexion réseau ou tenter de répondre aux requêtes de milliers de clients sur une machine multi-processeur.

Les ressources disponibles vont alors être extrêmement diverses et le serveur Internet devra s'adapter aux besoins du développeur et aux ressources disponibles. Pour chacune des architectures matérielles existantes, il existe un modèle de concurrence donné permettant d'exhiber les meilleures performances possibles et qui répond exactement aux demandes du développeur. Comme les serveurs Internet sont hébergés sur des machines hétérogènes, il est donc nécessaire d'adapter le plus facilement et rapidemment possible le modèle de concurrence en fonction de l'architecture matérielle sous-jacente afin de satisfaire le maximum de client en un minimum de temps et les besoins du développeur.

Du fait des nombreuses optimisations de code, de l'imbrication de la partie « métier » et des différentes préoccupations (dont la concurrence) le code source des serveurs Internet est difficilement lisible, maintenable et évolutif. C'est pourquoi pour adapter le modèle de concurrence à chaque architecture matérielle sous-jacente, il est nécessaire de redévelopper de « bout en bout » le même serveur Internet. Il y a donc un véritable manque de rationalisation en temps et en coût de développement [109, 111].

7.2 Contribution

J'ai réalisé une plateforme logicielle en Java qui offre une bonne illustration d'un nouveau style de programmation, les *fabriques ou usines d'applications* [89]. Cette plateforme, qui est dédiée aux serveurs Internet, est assimilable à une chaîne de production. Elle est basée sur trois concepts :

1. un graphe orienté de synchronisation et de communication ;
2. le code fonctionnel du serveur Internet ;
3. un modèle de concurrence choisi par le développeur.

La contribution majeure de mon travail est d'associer un ensemble de fonctionnalités provenant : (i) de la conception d'applications, (ii) des compilateurs, (iii) du développement d'applications et (iv) de la vérification formelle. Je ne propose aucune nouveauté dans l'une ou l'autre des techniques abordées, mais plutôt une unification de ces techniques dans une unique plateforme logicielle. De plus, l'unicité de la spécification assure la cohérence entre l'application et son modèle formel.

7.2.1 Génération du code concurrent des serveurs Internet

A partir de trois concepts cités ci-dessus, je suis capable de composer automatiquement un serveur Internet pleinement fonctionnel. L'approche que je propose permet une meilleure sûreté des logiciels produits. Elle permet d'éviter un grand nombre d'erreurs car le code concurrent traditionnellement sujet à de nombreux « comportements non désirés », est produit automatiquement. Elle offre aussi une grande portabilité et une prise en compte rapide de futures évolutions technologiques par simple ajout ou modification des règles de transformation du modèle de spécification vers les serveurs Internet produits. Enfin, elle offre une diminution du temps et des coûts de développement, ce qui permet aux développeurs de se

consacrer à d'autres fonctionnalités du serveur Internet et nécessite moins de connaissances techniques.

7.2.2 Génération du modèle formel d'un serveur Internet

Enfin lors de la composition des serveurs Internet, j'ai constaté qu'il y avait un besoin important en contrôle et en validation des assemblages. Bien que le graphe orienté de synchronisation et de communication y réponde partiellement de « manière déclarative » pour les assemblages prévus et que les interfaces de contrôles de mes générateurs tentent d'y répondre de « manière programmative ». Il est possible d'améliorer encore la sûreté des serveurs produit par Saburo.

J'ai ainsi proposé des générateurs d'abstraction sûre spécifiques à un domaine précis, les serveurs Internet [71]. Me focaliser sur un seul domaine m'a permis : (i) de ne pas avoir à spécifier les propriétés que je souhaite vérifier car elles dépendent bien souvent du domaine, (ii) simplifie le développement des générateurs et (iii) améliore la sûreté des abstractions vis-à-vis du système. Le modèle d'extraction que j'ai utilisé est un modèle descendant, c'est-à-dire que le modèle est généré à partir d'une meta-spécification. Il se base sur le graphe de spécification d'un serveur Internet et le modèle de concurrence choisi par le développeur. Le code des différents stages n'est pas nécessaire pour obtenir l'abstraction du serveur Internet car l'ensemble des opérations de synchronisation et de communication présentes dans un serveur Internet sont modélisées par le graphe de spécification. Les informations sélectionnées vont être suffisantes pour la vérification des propriétés principales d'un serveur Internet que sont (i) l'absence d'interblocage et (ii) l'atteignabilité de tous les états du système. Pour accroître la sûreté de l'application, il est possible d'*ajouter des propriétés supplémentaires à vérifier*, sous forme de formules de logiques temporelles. Pratiquement la phase de vérification est basée sur le *model checker* SPIN [47].

7.2.3 Génération de l'analyse syntaxique des requêtes clientes

Dans un second temps je me suis intéressé au problème du décodage des requêtes clientes. En effet, dans un serveur Internet, le décodage des requêtes est l'interface d'interaction entre le serveur et ses clients. La sûreté et la robustesse de ses traitements sont primordiaux lors de la conception d'un serveur car tout comportement non désiré peut permettre des accès illégaux au serveur ou empêcher de servir correctement un client. De plus, le décodage des requêtes clientes nécessite de prendre en compte le type d'E/S utilisées par le serveur. En effet si les E/S sont non bloquantes, il est nécessaire de sauvegarder le contexte du décodage, puis de le restaurer à chaque fois, ce qui n'est pas le cas pour des E/S bloquantes.

C'est pourquoi, j'ai utilisé *Tatoo*, un générateur d'analyseur syntaxique [23], qui m'a permis d'obtenir automatiquement des analyseurs de requêtes clientes [22]. Tatoo, contrairement aux principaux générateurs d'analyseurs actuels [36, 60, 90], permet de générer automatiquement des analyseurs qui sont compatibles avec des E/S bloquantes et non bloquantes. De plus, *Tatoo* a été conçu pour générer des analyseurs qui vont être embarqués dans des

serveurs Internet. Il prend en considération les contraintes de temps d'exécution et de gestion mémoire induites. J'ai réalisé des tests de performance pour un serveur HTTP qui m'ont permis de justifier pratiquement la faisabilité d'une telle approche. Ces tests ont été réalisés dans un contexte particulier de charge importante ce qui n'entraîne aucune pénalité pour les serveurs les plus complets.

7.3 Perspectives

Saburo permet de générer automatiquement le code concurrent des serveurs Internet ce qui permet un gain de temps important en temps et en coût de développement. Cependant, les serveurs Internet sont des applications qui doivent être performantes et nécessitent donc d'importantes optimisations. Actuellement, ces optimisations ne sont pas mises en œuvre dans tous les générateurs que je fournis. Les premiers travaux à effectuer concerneront l'enrichissement des générateurs afin d'offrir pour tous les modèles de concurrence les meilleures performances possibles.

Parallèlement, je souhaiterais faciliter l'utilisation et la diffusion de Saburo en développant un « greffon » Eclipse qui est une plateforme de développement très largement utilisée.

Concernant les tests de performances des serveurs Internet, je n'ai actuellement pas trouvé de logiciels qui permettent de simuler des coupures dans les requêtes qui sont envoyées au serveur. Je pense que ce phénomène permettra de mettre en exergue les capacités des serveurs Internet utilisant des analyseurs syntaxiques non bloquants à supporter un plus grand nombre de clients que les versions bloquantes.

Le modèle de développement que je propose permet aussi de développer des applications réparties. En effet, les connexions entre les différents stages vont se faire *via* des connexions réseaux. Je souhaite par la suite développer un peu plus cette thématique d'applications réparties. De plus, je pense que la configuration et le déploiement de ces applications sont deux choses extrêmement importantes pour faciliter et promouvoir l'utilisation de mon outil.

La spécification sous forme de graphe orienté est particulièrement bien adaptée : (i) à l'application d'algèbres de processus, (ii) à une transformation vers un réseau de pétri ou (iii) à la transformation vers un langage d'entrée d'un *model checker*. J'ai commencé avec succès quelques travaux préliminaires qui permettent de transformer le graphe de spécification de Saburo vers une expression en π-calculus. Je souhaite continuer dans cette voie et fournir un support vers d'autres langages de *model checker* comme Nu-SMV [24].

Actuellement, Saburo est un simple générateur de serveurs Internet. Je souhaiterais transformer Saburo en un compilateur de serveurs Internet. En effet et à la différence d'un générateur, un compilateur doit signaler à son utilisateur la présence d'erreurs dans le programme source. Du fait de l'utilisation du principe de tissage il existe une grande difficulté à déverminer une application. En effet, il est extrêmement difficile de savoir d'ou provient une erreur. De plus, une erreur peut seulement survenir lors de l'association de deux portions de code.

Bien qu'utilisant le concept de tissage, Saburo facilite le déverminage par une séparation claire entre : (i) la partie concurrence vérifiée à l'aide d'un *model checker* et (ii) la partie métier. Dans le modèle de développement que j'ai proposé, la partie métier présente des caractéristiques très particulières qui empêchent toute interaction avec la partie concurrence. Cependant, il va être nécessaire de faire le lien entre : (i) les sorties obtenues à l'aide des outils de vérification et (ii) le code en entrée fourni par le développeur. La partie métier, i.e. les différents stages, devra elle aussi être vérifiée dans le but de certifier l'absence de « comportements non désirés ».

Bibliographie

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management without Manual Stack Management. In *Proceedings of the USENIX Annual Technical Conference*, pages 289 – 302, Monterey, CA, USA, June 2002.

[2] A. Aho, R. Sethi, and R. Hullman. *Compilateurs : Principes, techniques et outils.* InterEditions, Paris, quatrième edition, 1991.

[3] M. Akil. Architecture des ordinateurs. Cours de DEA, 2004.

[4] T. Andrews, S. Qadeer, S. K. Rajamani, and Y. Xie. Zing : Exploiting Program Structure for Model Checking Concurrent Software. In *15th International Conference on Concurrency Theory*, volume 3170 of *LNCS*, pages 1–15, London, UK, August 2004.

[5] Apache Software Foundation. *Apache HTTP Server Benchmarking Tool.* `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[6] Apache Software Foundation. *The Apache Ant Project.* `http://ant.apache.org/`.

[7] Apache Software Foundation. *The Apache HTTP Server Project.* `http://httpd.apache.org/`.

[8] Apache Software Foundation. *Velocity 1.4 User Guide*, January 2006. `http://jakarta.apache.org/velocity/`.

[9] *Aspect-Oriented Programming for Java.* `http://aspectj.org`.

[10] G. Back. Datascript : A Specification and Scripting Language for Binary Data. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 66 – 77, London, UK, 2002. Springer-Verlag.

[11] R. Von Behren, J. Condit, and E. Brewer. Why Events are a Bad Idea (for High Concurrency Servers) ? In *Ninth Workshop on Hot Topics in Operating Systems*, pages 19 – 24, Lihue, Hawaï, USA, May 2003. USENIX, The Advanced Computing Systems Association.

[12] R. Von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio : Scalable Threads for Internet Services. In *Nineteenth ACM Symposium on Operating Systems Principles*, pages 268 – 281, Bolton Landing, NY, USA, October 2003. ACM Press.

[13] C. Bernardeschi, G. Dini, and A. Domenici. FACT : A tool for code generation from communicating automata. In *IASTED Conference on Software Engineering*, pages 313–318, Innsbruck, Austria, February 2005. ACTA Press.

[14] J. Berstel, S. C. Reghizzi, G. Roussel, and P. San Pietro. A scalable formal method for design and automatic checking of user interfaces. *ACM Transactions on Software Engineering Methodology*, 14(2) :124–167, 2005.

[15] J. Bézivin and X. Blanc. *MDA : Vers un important changement de paradigme en génie logiciel*, 2002. Développeur référence v2.16.

[16] N. Borisov, D. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. A Generic Application-Level Protocol Analyzer and its Language. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[17] N. M. N. Bouraqadi-Saâdani and T. Ledoux. Le Point sur la Programmation par Aspects. *Technique et Science Informatique*, 20(4) :505 – 528, December 2001.

[18] G. Bracha. Pluggable type systems. In *Object-Oriented Programming, Systems, Languages & Applications Workshop on Revival of Dynamic Languages*, 2004.

[19] G. W. Brams, C. Andre, G. Berthelot, C. Girault, G. Memmi, G. Roucairol, J. Sifakis, R. Valette, and G. Vidal-Naquet. *Reseaux de Petri : Theorie et Pratique.* Editions Masson, September 1982.

[20] E. Bruneton, R. Lenglet, and T. Coupaye. ASM : A Code Manipulation Tool for the Construction of Adaptable Systems. In *Adaptable and Extensible Component Systems*, Grenoble, France, November 2002. ACM Press.

[21] L. Burgy, L. Reveillere, J. L. Lawall, and G. Muller. A Language-Based Approach for Improving the Robustness of Network Application Protocol Implementations. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, pages 149 – 160, Washington, DC, USA, 2007. IEEE Computer Society.

[22] J. Cervelle, R. Forax, G. Loyauté, and G. Roussel. Banzaï : A java framework for the implementation of high-performance servers. In *Submited to Symposium on Applied Computing*, Honolulu, Hawaii, USA, March 2009.

[23] J. Cervelle, R. Forax, and G. Roussel. Tatoo : An innovative parser generator. In *International Conference on Principles and Practices of Programming In Java*, pages 13–20, Mannheim, Germany, August 2006. ACM International Conference Proceedings Series.

[24] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV : A New Symbolic Model Checker. *International Journal on Software Tools for Technology Transfer*, 2(4) :410 – 425, 2000.

[25] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244 – 263, 1986.

[26] J. Cocke. *Programming languages and their compilers : Preliminary notes.* Courant Institute of Mathematical Sciences, New York University, 1969.

[27] K. G. Coffman and A. M. Odlyzko. Internet growth : Is there a "moore's law" for data traffic ? In *Handbook of Massive Data Sets*, pages 47–93. Kluwer Academic Publishers, Norwell, MA, USA, June 2002.

[28] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, R., and Hongjun Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22th International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society.

[29] K. Czarnecki and U. W. Eisenecker. Components and generative programming (invited paper). In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 2 – 19, London, UK, 1999. Springer-Verlag.

[30] K. Czarnecki and U. W. Eisenecker. *Generative Programming, Methods, Tools and Applications*. Addison Wesley Professional, June 2000.

[31] A. Demenchuk. Beaver : A LALR Parser Generator, 2006. `http://beaver.sourceforge.net/index.html`.

[32] E. W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5) :341 – 346, 1968.

[33] M. Douglas. MSPL : A Protocol Language for Generating Client-Server Software. Master's thesis, Florida Tech, 2000.

[34] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2) :94 – 102, 1970.

[35] K. Fisher and R. Gruber. PADS : A Domain-Specific Language for Processing Ad Hoc Data. *SIGPLAN Notices*, 40(6) :295 – 304, 2005.

[36] E. M. Gagnon and L. J. Hendren. SableCC, an Object-Oriented Compiler Framework. In *Technology of Object-Oriented Languages and Systems*, pages 140 – 154. IEEE Computer Society, 1998.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Catalogue de modèles de conception réutilisables*. Vuibert Informatique, 1999.

[38] S. J. Garland and Nancy Lynch. Using I/O automata for developing distributed systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 13, pages 285–312. Cambridge Univ. Press, New York, USA, 2000.

[39] K. Gerwin. *JFlex User's Manual*, July 2005.

[40] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Professional, third edition, July 2005.

[41] D. Gupta and R. Jaiswal. Threads vs. Events. Report for CSE221, University of California, San Diego, USA, December 2003.

[42] P. Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7) :574 – 578, 1972.

[43] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. B. Trakhtenbrot. Statemate : A working environment for the development of complex reactive systems. In *Proceedings of the 10th International Conference*

on Software Engineering, pages 396–406, Los Alamitos, California, USA, April 1988. IEEE Computer Society.

[44] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4) :366–381, March 2000.

[45] Hewlett Packard. *Httperf*. `http://www.hpl.hp.com/research/linux/httperf/`.

[46] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8) :666 – 677, 1978.

[47] G. J. Holzmann. *The SPIN model checker : Primer and reference manual*. Addison Wesley Professional, September 2003.

[48] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[49] S. E. Hudson. *CUP User's Manual*. Usability Center, Georgia Institute of Technology, July 1999.

[50] W. Hürsch and C. Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, February 1995.

[51] International Organization for Standardization. *Open System Interconnection Model*, 1994.

[52] Internet Engineering Task Force. *HyperText Transfer Protocol – HTTP/1.1*, 1999.

[53] S. C. Johnson. Yacc : Yet Another Compiler Compiler, 1979.

[54] R. Jones and R. Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons Ltd, March 1996.

[55] J. Kasami. An Efficient Recognition and Syntax Analysis Algorithm for Context-Free Languages. Technical report, University of Hawaii, 1965.

[56] B. W. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall PTR, March 1988.

[57] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace : Language Support for Building Distributed Systems. *SIGPLAN Notice*, 42(6) :179 – 188, 2007.

[58] J. S. Kim and D. Garlan. Analyzing Architectural Styles with Alloy. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 70 – 80, New York, NY, USA, 2006. ACM Press.

[59] A. Knapp and S. Merz. Model checking and code generation for UML state machines and collaborations. In *5th Tools for System Design and Verification*, Günzburg, Germany, July 2002.

[60] V. Kodaganallur. Incorporating Language Processing into Java Applications : A JavaCC Tutorial. *IEEE Software*, 21(4) :70 – 77, August 2004.

[61] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3) :872–923, May 1994.

[62] H. C. Lauer and R. M. Needham. On the Duality of Operating System Structures. *SIGOPS Operating Systems Review*, 13(2) :3 – 19, April 1979. Reprint of the original paper by H. C. Lauer and R. M. Needham in 'Proceedings of the Second Symposium on Operating Systems.

[63] D. Lea. *Concurrent programming in Java : Design principles and pattern.* The Java Series. Addison Wesley Professional, second edition, October 1999.

[64] W. LeFebvre. Cnn.com : Facing a world crisis. In *Invited talk at USENIX LISA*, , USA, December 2001. USENIX Association.

[65] M. E. Lesk and E. Schmidt. Lex : A Lexical Analyzer Generator. Technical Report 39, Bell Laboratories, July 1975.

[66] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 2nd edition, 1999.

[67] D. T. Lowell. *APG : An ABNF Parser Generator*, June 2006. `http://www.coasttocoastresearch.com/`.

[68] G. Loyauté. A framework for development of concurrency and i/o in servers. In *Poster session of the First European Conference on System (EuroSys06)*, Leuven, Belgium, April 2006.

[69] G. Loyauté. Multi single-process event-driven : A new internet server architecture. In *Poster session of the Second European Conference on System (EuroSys07)*, Lisbon, Portugal, March 2007.

[70] G. Loyauté, R. Forax, and G. Roussel. Saburo : a tool for I/O and concurrency management in servers. In *Eighteen International Workshop on Java for Parallel and Distributed Computing (JavaPDC'06) held in conjunction with 20th International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, April 2006. IEEE Computer Society.

[71] G. Loyauté, R. Forax, and G. Roussel. A Java Method for the Design and the Automatic Checking of Server Architectures. In *International Conference on the Principles and Practices of Programming in Java*, Lisbon, Portugal, September 2007.

[72] J. Magee, N. Dulay, and J. Kramer. Regis : A Constructive Development Environment for Distributed Programs. *Distributed Systems Engineering*, 1(5) :304–312, 1994.

[73] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour Analysis of Software Architectures. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture*, pages 35–50, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[74] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[75] N. Markey and P. Schnoebelen. TSMV : A Symbolic Model Checker for Quantitative Analysis of Systems. In *QEST '04 : Proceedings of the The Quantitative Evaluation of Systems, First International Conference*, pages 330 – 331, Washington, DC, USA, 2004. IEEE Computer Society.

[76] R. Marvie and M.-C. Pellegrini. Modèles de composants, un état de l'art. *Numéro spécial de l'Objet*, 8(3), 2002.

[77] P. J. McCann and S. Chandra. Packet Types : Abstract Specification of Network Protocol Messages. *SIGCOMM Computer Communication Review*, 30(4) :321 – 333, 2000.

[78] Microsoft. *.NET Remoting Overview.* `http://msdn2.microsoft.com/en-us/library/kwdt6w2k(VS.71).aspx`.

[79] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[80] R. Milner. *Communicating and Mobile Systems : the Pi-Calculus.* Cambridge University Press, fifth edition, 2004.

[81] Mort Bay Consulting. *Jetty.* `http://www.mortbay.org/`.

[82] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *International Symposium on Software Testing and Analysis*, pages 229–239. ACM Press, 2002.

[83] The Object Management Group. `http://www.omg.org`.

[84] The Object Management Group. *Unified Modeling Language.* `http://www.uml.org`.

[85] J. Ousterhout. Why Threads Are a Bad Idea (for most purposes). In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, USA, January 1996. Invited talk.

[86] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash : An efficient and portable web server. In *USENIX Annual Technical Conference*, pages 199–212, Monterey, CA, USA, June 1999.

[87] R. Pang, V. Paxson, R. Sommer, and L. Peterson. Binpac : A Yacc for Writing Application Protocol Parsers. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, pages 289 – 300, New York, NY, USA, 2006. ACM.

[88] D. Pariag, T. Brecht, A. Harji, P. Buhr, and A. Shukla. Comparing the Performance of Web Server Architectures. In *Second European Conference on System (EuroSys07)*, pages –, Lisbon, Portugal, March 2007.

[89] D. Parigot. Contribution à la programmation générative. Habilitation à diriger des recherches, 2003.

[90] T. J. Parr and R. W. Quong. ANTLR : A Predicated-LL(k) Parser Generator. *Software Practice and Experience*, 25(7) :789 – 810, 1995.

[91] R. Pawlak. Spoon : Annotation-Driven Program Transformation — The AOP Case. In *Proceedings of the 1st workshop on Aspect oriented middleware development*, New York, NY, USA, 2005. ACM.

[92] G. L. Peterson. A New Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables. *ACM Transactions on Programming Languages and Systems*, 5(1) :56 – 65, 1983.

[93] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[94] C.A. Petri. Communication with Automata. Technical Report RADC-TR-65-377, Griffiss Air Force Base, New York, USA, 1966.

[95] X. Qie, R. Pang, and L. Peterson. Defensive Programming : Using an Annotation Toolkit to Build DoS-Resistant Software. In *Proceedings of the 5th symposium on Operating Systems Design and Implementation*, pages 45 – 60, New York, NY, USA, 2002. ACM.

[96] Wolfgang Reisig. *Petri Nets : An Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[97] G. Roussel, E. Duris, N. Bedon, and R. Forax. *Java et Internet : Concepts et Programmation*. Vuibert Informatique, 2002.

[98] A. Sakharov. State machine specification directly in Java and C++. In *Poster to the proceedings of Object-Oriented Programming, Systems, Languages & Applications*, pages 103–104. ACM Press, 2000.

[99] The SPEC Consortium. *SPECweb2005*. `http://www.spec.org/web2005/`.

[100] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, second edition, February 2000.

[101] Sun Microsystems. *Enterprise Java Beans*. `http://java.sun.com/products/ejb`.

[102] Sun Microsystems. *GlassFish » Grizzly Project*. `https://grizzly.dev.java.net/`.

[103] Sun Microsystems. *Mécanisme de réflexion en Java*. `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/reflect/package-summary.html`.

[104] Sun Microsystems. *New I/O APIs*. `http://java.sun.com/j2se/1.4.2/docs/guide/nio/`.

[105] Sun Microsystems, Mountain View, California, USA. *JavaBeans API Specification*, August 1997.

[106] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison Wesley Professional, 1998.

[107] The Apache Software Foundation. *Apache Tomcat 6.0*. `http://tomcat.apache.org/tomcat-6.0-doc/index.html`.

[108] The Object Management Group. *The Corba Component Model*. `http://www.omg.org`.

[109] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA : An architecture for well-conditioned, scalable Internet services. In *Eighteenth Symposium on Operating Systems Principles*, pages 230 – 243, Chateau Lake Louise, Canada, October 2001. ACM Press.

[110] J. Wilcox. Users say msn outage continues. In *ZDNet News*. July 2001.

[111] K. Wilson and J. Aycok. NEST : NEtwork Server Tool. Technical Report TR-2004-746-11, The University of Calgary, April 2004.

[112] The World Wide Web Consortium. *http://www.w3.org*.

[113] The World Wide Web Consortium. *Web Services Activity*. `http://www.w3.org/2002/ws/`.

[114] N.-M. Yao, M.-Y. Zheng, and J.-B. Ju. Pipeline : A new architecture of high-performance servers. *SIGOPS Operating Systems Review*, 36(4) :55–64, 2002.

[115] D. H. Younger. Recognition and parsing of context-free languages in time n3. *nformation and Control*, 10 :189 – 208, 1967.

[116] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazieres, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the USENIX Annual Technical Conference*, pages 239 – 252, San Antonio, Texas, USA, June 2003.

A

Abstraction en Promela d'un serveur HTTP

A.1 Transcription en Promela du modèle itératif

```
#define MAX_MESSAGE 10
#define MAX_CLIENT 5

mtype = { REQUEST, RESPONSE, CLOSE, NULL };
mtype = { OP_ACCEPT, OK_ACCEPT, NO_ACCEPT };
mtype = { OP_READ, OK_READ, NO_READ };
mtype = { OP_DECODE, OK_DECODE, NO_DECODE };
mtype = { OP_SERVICE, OK_SERVICE, NO_SERVICE };
mtype = { OP_ENCODE, OK_ENCODE, NO_ENCODE };
mtype = { OP_WRITE, OK_WRITE, NO_WRITE };

chan server = [ MAX_MESSAGE ] of { mtype, chan };

active proctype IterativeHttp() {
    chan clt[ 1 ] = [ 0 ] of { mtype, chan };
    chan lock = [ 1 ] of { chan };

    chan internal = [ MAX_MESSAGE ] of { mtype, chan, chan };

    chan s_write;
    chan s_read;

    byte i;
    start : atomic {
      do
      : : i < 1 ->
        lock !clt[ i ];
        i++;
      : : else ->
        break;
      od;
    }

    end : do
      : : server ?< REQUEST, s_write > ->
        if
        : : nempty( lock ) ->
```

```
      server?REQUEST( s_write );
      lock?s_read;
      internal!OP_ACCEPT( s_write, s_read );
      goto end;
    : : empty( lock ) ->
      goto end;
    fi;
  : : internal?OP_ACCEPT( s_write, s_read ) ->
    s_write!OK_ACCEPT( s_read );
    internal!OP_READ( s_write, s_read );
    goto end;
  : : internal?OP_READ( s_write, s_read ) ->
    if
    : : s_write!OK_READ( s_read ) ->
      if
      : : s_read?OK_READ( s_write ) ->
        internal!OP_DECODE( s_write, s_read );
        goto end;
      : : s_read?NO_READ( s_write ) ->
        goto release;
      fi;
    : : s_write!NO_READ( s_read ) ->
      printf("Server IOException!!!");
      goto release;
    fi;
  : : internal?OP_DECODE( s_write, s_read ) ->
    if
    : : internal!OP_SERVICE( s_read ) ->
      goto end;
    : : skip ->
      goto release;
    fi;
  : : internal?OP_SERVICE( s_write, s_read ) ->
    if
    : : internal!OP_ENCODE( s_read ) ->
      goto end;
    : : skip ->
      goto release;
    fi;
  : : internal?OP_ENCODE( s_write, s_read ) ->
    if
    : : internal!OP_WRITE( s_read ) ->
      goto end;
    : : skip ->
```

```
          goto release;
        fi;
      : : internal?OP_WRITE( s_write, s_read ) ->
        if
        : : s_write!OK_WRITE( s_read ) ->
          if
          : : s_read?OK_WRITE( s_write ) ->
            s_write!RESPONSE( s_read );
            s_read?CLOSE( s_write );
            goto release;
          : : s_read?NO_WRITE( s_write ) ->
            goto release;
          fi;            : : s_write!NO_WRITE( s_read ) ->
            printf("Server IOException!!!");
            goto release;
          fi;
    release : lock!s_read;
    od;
}
```

A.2 Transcription en Promela du modèle SPED

```
#define MAX_MESSAGE 10
#define MAX_CLIENT 5

mtype = { REQUEST, RESPONSE, CLOSE, NULL };
mtype = { OP_ACCEPT, OK_ACCEPT, NO_ACCEPT };
mtype = { OP_READ, OK_READ, NO_READ };
mtype = { OP_DECODE, OK_DECODE, NO_DECODE };
mtype = { OP_SERVICE, OK_SERVICE, NO_SERVICE };
mtype = { OP_ENCODE, OK_ENCODE, NO_ENCODE };
mtype = { OP_WRITE, OK_WRITE, NO_WRITE };

chan server = [ MAX_MESSAGE ] of { mtype, chan };

active proctype SpedHttp() {
    chan clt[ MAX_CLIENT ] = [ 0 ] of { mtype, chan };
    chan lock = [ MAX_CLIENT ] of { chan };

    chan internal = [ MAX_MESSAGE ] of { mtype, chan, chan };

    chan s_write;
    chan s_read;

    byte i;

    start : atomic {
      do
      : : i < MAX_CLIENT ->
        lock!clt[ i ];
        i++;
      : : else ->
        break;
      od;
```

```
    }

end : do
    : : server ?< REQUEST, s_write > ->

        if
      : : nempty( lock ) ->
        server ?REQUEST( s_write ) ;
        lock ?s_read ;
        internal !OP_ACCEPT( s_write, s_read ) ;
        goto end ;
      : : empty( lock ) ->
        goto end ;
      fi ;
    : : internal ?OP_ACCEPT( s_write, s_read ) ->
      s_write !OK_ACCEPT( s_read ) ;
      internal !OP_READ( s_write, s_read ) ;
      goto end ;
    : : internal ?OP_READ( s_write, s_read ) ->
      if
      : : s_write !OK_READ( s_read ) ->
        if
        : : s_read ?OK_READ( s_write ) ->
          internal !OP_DECODE( s_write, s_read ) ;
          goto end ;
        : : s_read ?NO_READ( s_write ) ->
          goto release ;
        fi ;
      : : s_write !NO_READ( s_read ) ->
        goto release ;
      fi ;
    : : internal ?OP_DECODE( s_write, s_read ) ->
      if
      : : internal !OP_SERVICE( s_read ) ->
        goto end ;
      : : skip ->
        goto release ;
      fi ;
    : : internal ?OP_SERVICE( s_write, s_read ) ->
      if
      : : internal !OP_ENCODE( s_read ) ->
        goto end ;
      : : skip ->
        goto release ;
      fi ;
    : : internal ?OP_ENCODE( s_write, s_read ) ->
```

```
      if
      : : internal !OP_WRITE( s_read ) ->
        goto end ;
      : : skip ->
        goto release ;
      fi ;
    : : internal ?OP_WRITE( s_write, s_read ) ->
    if
      : : s_write !OK_WRITE( s_read ) ->
        if
        : : s_read ?OK_WRITE( s_write ) ->
          s_write !RESPONSE( s_read ) ;
          s_read ?CLOSE( s_write ) ;
          goto release ;
        : : s_read ?NO_WRITE( s_write ) ->
          goto release ;
        fi ;
      : : s_write !NO_WRITE( s_read ) ->
        goto release ;
      fi ;
release : lock !s_read ;
    od ;
}
```

A.3 Transcription en Promela du modèle multi-processus légers

```
#define MAX_MESSAGE 10
#define MAX_CLIENT 5
#define MAX_THREAD 5

mtype = { REQUEST, RESPONSE, CLOSE, NULL } ;
mtype = { OP_ACCEPT, OK_ACCEPT, NO_ACCEPT } ;
mtype = { OP_READ, OK_READ, NO_READ } ;
mtype = { OP_DECODE, OK_DECODE, NO_DECODE } ;
mtype = { OP_SERVICE, OK_SERVICE, NO_SERVICE } ;
mtype = { OP_ENCODE, OK_ENCODE, NO_ENCODE } ;
mtype = { OP_WRITE, OK_WRITE, NO_WRITE } ;

chan server = [ MAX_MESSAGE ] of { mtype, chan } ;

chan ipc = [ MAX_MESSAGE ] of { mtype, chan } ;

proctype HttpAgent( chan s_write, s_read ) {
    chan internal = [ MAX_MESSAGE ] of { mtype, chan, chan } ;

    internal !OP_READ( s_write, s_read ) ;

    end : do
    : : internal ?OP_READ( s_write, s_read ) ->
      if
      : : s_write !OK_READ( s_read ) ->
```

```
        if
        :: s_read?OK_READ( s_write ) ->
          internal!OP_DECODE( s_write, s_read );
          goto end;
        :: s_read?NO_READ( s_write ) ->
          goto release;
        fi;
      :: s_write!NO_READ( s_read ) ->
        printf("Server IOException!!!");
        goto release;
      fi;
    :: internal?OP_DECODE( s_write, s_read ) ->
      if
      :: internal!OP_SERVICE( s_read ) ->
        goto end;
      :: skip ->
        goto release;
      fi;
    :: internal?OP_SERVICE( s_write, s_read ) ->
      if
      :: internal!OP_ENCODE( s_read ) ->
        goto end;
      :: skip ->
        goto release;
      fi;
    :: internal?OP_ENCODE( s_write, s_read ) ->
      if
      :: internal!OP_WRITE( s_read ) ->
        goto end;
      :: skip ->
        goto release;
      fi;
    :: internal?OP_WRITE( s_write, s_read ) ->
      if
      :: s_write!OK_WRITE( s_read ) ->
        if
        :: s_read?OK_WRITE( s_write ) ->
          s_write!RESPONSE( s_read );
          s_read?CLOSE( s_write );
          goto release;
        :: s_read?NO_WRITE( s_write ) ->
          goto release;
        fi;      :: s_write!NO_WRITE( s_read ) ->
        printf("Server IOException!!!");
        goto release;
      fi;
    od;

    release : ipc!RELEASE( s_read );
}

active proctype MultithreadedHttp() {
    chan clt[ MAX_THREAD ] = [ 0 ] of { mtype, chan };

    chan lock = [ MAX_THREAD ] of { chan };

    chan internal = [ MAX_MESSAGE ] of { mtype, chan, chan };

    chan s_write;
    chan s_read;

    byte i;

    start : atomic {
      do
      :: i < MAX_THREAD ->
        lock!clt[ i ];
```

```
        i++;
      : : else ->
        break;
      od;      }

    end : do
    : : server?< REQUEST, s_write > ->
      if
      : : nempty( lock ) ->
        server?REQUEST( s_write );
        lock?s_read;
        internal!OP_ACCEPT( s_write, s_read );
        goto end;
      : : empty( lock ) ->
        goto end;        fi;
    : : internal?OP_ACCEPT( s_write, s_read ) ->
      s_write!OK_ACCEPT( s_read );
      run EchoAgent( s_write, s_read );
      goto end;
    : : ipc?RELEASE( s_read ) ->
      lock!s_read;
    od;
}
```

A.4 Transcription en Promela du modèle pipeline

```
#define MAX_MESSAGE 10
#define MAX_CLIENT 5
#define MAX_POOL 5

mtype = { REQUEST, RESPONSE, CLOSE, NULL };
mtype = { OP_ACCEPT, OK_ACCEPT, NO_ACCEPT };
mtype = { OP_READ, OK_READ, NO_READ };
mtype = { OP_DECODE, OK_DECODE, NO_DECODE };
mtype = { OP_SERVICE, OK_SERVICE, NO_SERVICE };
mtype = { OP_ENCODE, OK_ENCODE, NO_ENCODE };
mtype = { OP_WRITE, OK_WRITE, NO_WRITE };

chan server = [ MAX_MESSAGE ] of { mtype, chan };

chan error = [ MAX_MESSAGE ] of { mtype, chan };

chan acceptToRead = [ MAX_MESSAGE ] of { mtype, chan, chan };
chan readToDecode = [ MAX_MESSAGE ] of { mtype, chan, chan };
chan decodeToService = [ MAX_MESSAGE ] of { mtype, chan, chan };
chan serviceToEncode = [ MAX_MESSAGE ] of { mtype, chan, chan };
chan encodeToWrite = [ MAX_MESSAGE ] of { mtype, chan, chan };

active proctype ReadStage() {
    chan s_read;
    chan s_write;

    end : do
    : : acceptToRead?OP_READ( s_write, s_read ) ->
      if
```

```
      : : s_write !OK_READ( s_read ) ->
      if
      : : s_read ?OK_READ( s_write ) ->
        readToDecode !OP_DECODE( s_write, s_read ) ;
        goto end ;
      : : s_read ?NO_READ( s_write ) ->
        goto release ;
      fi ;
      : : s_write !NO_READ( s_read ) ->
        printf("Server IOException ! ! !") ;
        goto release ;
      fi ;
      od ;

    release : {
      error !RELEASE( s_read ) ;
      goto end ;
    }
}

active proctype DecodeStage() {
    chan s_read ;
    chan s_write ;

    end : do
    : : readToDecode ?OP_DECODE( s_write, s_read ) ->
      if
      : : decodeToService !OP_SERVICE( s_read ) ->
        goto end ;
      : : skip ->
        goto release ;
      fi ;
    od ;

    release : {
      error !RELEASE( s_read ) ;
      goto end ;
    }
}

active proctype ServiceStage() {
    chan s_read ;
    chan s_write ;

    end : do
    : : decodeToService ?OP_SERVICE( s_write, s_read ) ->
      if
      : : serviceToEncode !OP_ENCODE( s_read ) ->
        goto end ;
      : : skip ->
        goto release ;
      fi ;
    od ;

    release : {
      error !RELEASE( s_read ) ;
      goto end ;
    }
}

active proctype EncodeStage() {
    chan s_read ;
    chan s_write ;

    end : do
    : : serviceToEncode ?OP_ENCODE( s_write, s_read ) ->
      if
```

```
      : : encodeToWrite!OP_WRITE( s_read ) ->
        goto end;
      : : skip ->
        goto release;
      fi;
    od;

    release : {
      error!RELEASE( s_read );
      goto end;
    }
}

active proctype WriteStage() {
    chan s_read;
    chan s_write;

    end : do
    : : encodeToWrite?OP_WRITE( s_write, s_read ) ->
      if
      : : s_write!OK_WRITE( s_read ) ->
      if
      : : s_read?OK_WRITE( s_write ) ->
        s_write!RESPONSE( s_read );
        s_read?CLOSE( s_write );
        goto release;
      : : s_read?NO_WRITE( s_write ) ->
        goto release;
      fi;
      : : s_write!NO_WRITE( s_read ) ->
        printf("Server IOException!!!");
        goto release;
      fi;
    od;

    release : {
      error!RELEASE( s_read );
      goto end;
    }
}

active proctype PipelineHttp() {
    chan internal = [ MAX_MESSAGE ] of { mtype, chan, chan };

        chan clt[ MAX_POOL ] = [ 0 ] of { mtype, chan };
    chan pool = [ MAX_POOL ] of { chan };

    byte i;

    chan s_read;
    chan s_write;

        start : atomic {
      do
      : : i < MAX_POOL ->
        pool!clt[ i ];
        i++;
      : : else ->
        break;
      od;
    }


    end : do
    : : server?< REQUEST, s_write > ->
      if
```

```
      : : nempty( pool ) ->
        server?REQUEST( s_write );
        pool?s_read;
        internal!OP_ACCEPT( s_write, s_read );
        goto end;
      : : empty( pool ) ->
        goto end;       fi;
    : : internal?OP_ACCEPT( s_write, s_read ) ->
      s_write!OK_ACCEPT( s_read );
      acceptToRead!OP_READ( s_write, s_read );
      goto end;
    : : error?RELEASE( s_read ) ->
      pool!s_read;
    od;
}
```

A.5 Transcription en Promela du modèle SEDA

```
#define MAX_MESSAGE 10
#define MAX_CLIENT 5

mtype = { REQUEST, RESPONSE, CLOSE, NULL };
mtype = { OP_ACCEPT, OK_ACCEPT, NO_ACCEPT };
mtype = { OP_READ, OK_READ, NO_READ };
mtype = { OP_DECODE, OK_DECODE, NO_DECODE };
mtype = { OP_SERVICE, OK_SERVICE, NO_SERVICE };
mtype = { OP_ENCODE, OK_ENCODE, NO_ENCODE };
mtype = { OP_WRITE, OK_WRITE, NO_WRITE };

chan server = [ MAX_MESSAGE ] of { mtype, chan };

chan error = [ MAX_MESSAGE ] of { mtype, chan };

chan acceptToRead = [ MAX_MESSAGE ] of { mtype, chan, chan };
chan readToDecode = [ MAX_MESSAGE ] of { mtype, chan, chan };
chan decodeToService = [ MAX_MESSAGE ] of { mtype, chan, chan };
chan serviceToEncode = [ MAX_MESSAGE ] of { mtype, chan, chan };
chan encodeToWrite = [ MAX_MESSAGE ] of { mtype, chan, chan };

active proctype ReadStage() {
    chan s_read;
    chan s_write;

    end : do
    : : acceptToRead?OP_READ( s_write, s_read ) ->
      if
      : : s_write!OK_READ( s_read ) ->
      if
      : : s_read?OK_READ( s_write ) ->
        readToDecode!OP_DECODE( s_write, s_read );
        goto end;
      : : s_read?NO_READ( s_write ) ->
        goto release;
      fi;
      : : s_write!NO_READ( s_read ) ->
        printf("Server IOException!!!");
        goto release;
```

```
      fi;
      od;

    release : {
      error!RELEASE( s_read );
      goto end;
    }
}

active proctype DecodeStage() {
    chan s_read;
    chan s_write;

    end : do
    : : readToDecode?OP_DECODE( s_write, s_read ) ->
      if
      : : decodeToService!OP_SERVICE( s_read ) ->
        goto end;
      : : skip ->
        goto release;
      fi;
    od;

    release : {
      error!RELEASE( s_read );
      goto end;
    }
}

active proctype ServiceStage() {
    chan s_read;
    chan s_write;

    end : do
    : : decodeToService?OP_SERVICE( s_write, s_read ) ->
      if
      : : serviceToEncode!OP_ENCODE( s_read ) ->
        goto end;
      : : skip ->
        goto release;
      fi;
    od;

    release : {
      error!RELEASE( s_read );
      goto end;
    }
}

active proctype EncodeStage() {
    chan s_read;
    chan s_write;

    end : do
    : : serviceToEncode?OP_ENCODE( s_write, s_read ) ->
      if
      : : encodeToWrite!OP_WRITE( s_read ) ->
        goto end;
      : : skip ->
        goto release;
      fi;
    od;

    release : {
      error!RELEASE( s_read );
      goto end;
    }
```

```
}

active proctype WriteStage() {
    chan s_read;
    chan s_write;

    end : do
    : : encodeToWrite?OP_WRITE( s_write, s_read ) ->
      if
      : : s_write!OK_WRITE( s_read ) ->
      if
      : : s_read?OK_WRITE( s_write ) ->
        s_write!RESPONSE( s_read );
        s_read?CLOSE( s_write );
        goto release;
      : : s_read?NO_WRITE( s_write ) ->
        goto release;
      fi;
      : : s_write!NO_WRITE( s_read ) ->
        printf("Server IOException!!!");
        goto release;
      fi;
    od;

    release : {
      error!RELEASE( s_read );
      goto end;
    }
}

active proctype SedaHttp() {
    chan internal = [ MAX_MESSAGE ] of { mtype, chan, chan };

    chan clt[ MAX_CLIENT ] = [ 0 ] of { mtype, chan };
    chan pool = [ MAX_CLIENT ] of { chan };

    byte i;

    chan s_read;
    chan s_write;

    start : atomic {
      do
      : : i < MAX_CLIENT ->
        pool!clt[ i ];
        i++;
      : : else ->
        break;
      od;
    }


    end : do
    : : server?< REQUEST, s_write > ->
      if
      : : nempty( pool ) ->
        server?REQUEST( s_write );
        pool?s_read;
        internal!OP_ACCEPT( s_write, s_read );
        goto end;
      : : empty( pool ) ->
        goto end;
```

```
      fi;
    : : internal?OP_ACCEPT( s_write, s_read ) ->
      s_write!OK_ACCEPT( s_read );
      acceptToRead!OP_READ( s_write, s_read );
      goto end;
    : : error?RELEASE( s_read ) ->
      pool!s_read;
    od;
}
```

B.

Génération de l'analyse syntaxique

B.1 L'interface GrammarEvaluator

```
public interface GrammarEvaluator {
   /**
   * This methods is called after the reduction of the non
   * terminal request by the grammar production request.
   */
   public void request() ;

   /**
   * This methods is called after the reduction of the non
   * terminal firstline by the grammar production firstline.
   */
   public void firstline(Method m, String url, Version v) ;

   /**
   * This methods is called after the reduction of the non
   * terminal method by the grammar production method_get.
   */
   public Method method_get(Method get) ;

   /**
   * This methods is called after the reduction of the non
   * terminal version by the grammar production version_http11.
   */
   public Version version_http11(Version http11) ;

   /**
   * This methods is called after the reduction of the non
   * terminal version by the grammar production version_http10.
   */
   public Version version_http10(Version http10) ;

   /**
   * This methods is called after the reduction of the non
   * terminal version by the grammar production version_http09.
   */
   public Version version_http09(Version http09) ;

   /**
   * This methods is called after the reduction of the non
   * terminal header_part by the grammar production header_part.
   */
   public void header_part(String key, String value) ;
```

```
    /**
    * This methods is called after the reduction of the non
    * terminal endline by the grammar production endline.
    */
    public void endline();
}
```

B.2 L'interface TerminalEvaluator

```
/**
* @param <D> data type passed by the lexer listener.
*/
public interface TerminalEvaluator<D> {
    /**
    * This method is called when the rule <code>get</code> is
    * recognized by the lexer.
    */
    public Method get(D data);

    /**
    * This method is called when the rule <code>url</code> is
    * recognized by the lexer.
    */
    public String url(D data);

    /**
    * This method is called when the rule <code>http11</code> is
    * recognized by the lexer.
    */
    public Version http11(D data);

    /**
    * This method is called when the rule <code>http10</code> is
    * recognized by the lexer.
    */
    public Version http10(D data);

    /**
    * This method is called when the rule <code>http09</code> is
    * recognized by the lexer.
    */
    public Version http09(D data);

    /**
    * This method is called when the rule <code>header_key</code> is
    * recognized by the lexer.
    */
    public String header_key(D data);

    /**
    * This method is called when the rule <code>header_value</code> is
    * recognized by the lexer.
    */
    public String header_value(D data);
}
```